ANIMA: REPRESENTANDO E INTEGRANDO OBJETOS EDUCACIONAIS NA *WEB*

ANDRÉ SANTANCHÈ

Salvador

Abril – 2002

UNIVERSIDADE SALVADOR

PROGRAMA DE PÓS-GRADUAÇÃO EM REDES DE COMPUTADORES

ANIMA: REPRESENTANDO E INTEGRANDO OBJETOS EDUCACIONAIS NA *WEB*

André Santanchè

Dissertação apresentada à Universidade Salvador, como parte das exigências do Curso de Mestrado Profissional em Redes de Computadores, área de concentração em Redes de Computadores, para obtenção do título de "Mestre".

Orientador

Prof. Dr. Cesar Augusto Camillo Teixeira

Salvador
Abril – 2002

Ficha Catalográfica – Sistema de Bibliotecas da Unifacs

Santanchè, André,

ANIMA : representando e integrando objetos educacionais na WEB / André Santanchè ; orientador Prof. Dr. Cesar Augusto Camillo Teixeira. - Salvador : UNIFACS/Programa de Pós-Graduação em Redes de Computadores, 2002.

x, 153 f. : il., col.

Inclui referência bibliográfica e apêndice.

Dissertação apresentada à Universidade Salvador, como parte das exigências do Curso de Mestrado Profissional em Redes de Computadores para obtenção do título de Mestre.

1.Sistemas de Computação – Modelos – Educação. I. Teixeira, Cesar Augusto Camillo, orient. II.UNIFACS Programa de Pós-Graduação em Redes de Computadores. III. Título.

CDD: 003.3071

Para Mone,
eterna namorada.

Para Lys,
exemplo de perseverança.

Para Fancesco,
fonte de criatividade.

Para Pepe, Giulio, Ciça e Ia,
sempre manos e amigos

AGRADECIMENTOS

Ao meu orientador e amigo, Cesar Teixeira, pela paciência com minhas idéias mirabolantes e pela capacidade de orientar e direcionar meu trabalho com dedicação e seriedade.

A minha esposa Simone, que sempre me incentivou, apoiou, compreendeu e teve paciência nas muitas noites e fins de semana que fiquei diante da tela de um computador.

A minha mãe Lys, que me ajudou muito e teve dedicação e paciência de revisar o texto “duzentas vezes”.

Aos meus irmãos Pepe, Giulio, Ciça e Ia; familiares e amigos pelo constante incentivo e por sempre acreditarem no meu potencial.

À UNIFACS, especialmente ao Prof. Manoel Barros, pelo crédito e apoio que deu durante todos estes anos a minha pesquisa.

A Eliana Brenner, que sempre me incentivou a ser um autêntico pesquisador.

Aos meus companheiros e amigos do Volare. À Carol e à Chris, que junto comigo deram os primeiros passos nas pesquisas.

Aos meus colegas e amigos do NUPPEAD Clarissa, Delanise e Teresinha, pelo constante apoio, pelas longas conversas e debates, que foram muito importantes para o meu trabalho.

Aos colegas de mestrado e do NUPERC, companheiros de batalha e amigos. A Jorge Campos, amigo com quem troquei as primeiras idéias desta pesquisa.

Aos colegas do Anchieta, pelo apoio e pela amizade preciosa e aos professores que foram matéria prima da minha pesquisa. À Márcia, pela paciência em testar e re-testar o sistema Casa Mágica e a Reginaldo, Hernani e à Tonha, grandes colegas.

Aos amigos e irmãos do Focolare, pelo constante apoio, compreensão e orações nos momentos difíceis.

À Leila, que sempre teve paciência com meus atrasos e esquecimentos na pós-graduação e sempre me auxiliou em várias etapas do meu trabalho.

Às várias pessoas que testaram e utilizaram o sistema Casa Mágica, pela paciência e pelas preciosas sugestões.

Aos meus alunos, pela paciência nos momentos em que estive atarefado com a dissertação.

A todos aqueles que não mencionei, não menos importantes, muito obrigado!

“Que aproveita ao homem ganhar o mundo inteiro se perder a sua alma?”
(Mc 8, 36)

RESUMO

Santanchè, A. **Anima: Representando e Integrando Objetos Educacionais na *Web***. 2002. Dissertação (Mestrado Profissional em Redes de Computadores), Universidade Salvador, Salvador.

Palavras-chave: Objetos Educacionais, Componentes Educacionais, XML, RDF, Casa Mágica.

A combinação de modernas linguagens de código móvel, especialmente aquelas orientadas a objetos, com novos padrões de marcação estruturada, abertos e extensíveis – tais como XML e suas derivações – tem traçado novos caminhos na produção de aplicações *Web*.

Esta dissertação aborda como tecnologias emergentes na *Web* foram combinadas para estruturar um projeto, denominado Projeto *Anima*, cujo objetivo é integrar objetos de diferentes linguagens e implementações, principalmente aquelas de código móvel, em aplicações educacionais. Como resultado, obtemos conjuntos intercambiáveis de objetos capazes de explorar os melhores recursos de cada sistema, de ser combinados em produtos unificados, de trafegar pela *Web* de forma independente e de proporcionar maior integração e colaboração entre desenvolvedores.

ABSTRACT

Santanchè, A. ***Anima*: Representing and Integrating Educational Objects on the *Web***. 2002. Thesis (Degree of Master in Computer Networks), Universidade Salvador, Salvador.

Keywords: Educational Objects, Educational Components, XML, RDF, Casa Mágica.

The combination of modern mobile code languages, especially the object-oriented languages, with new open and extensible structured markup standards – as XML and their derivations – has designed new ways for the production of Web applications.

The main approach of this dissertation concerns how Web emergent technologies are combined in order to structure a project, referred to as the *Anima* project, of which the purpose is integrating objects of different languages and implementations, especially those of mobile code, in educational applications. As a result, we get interchangeable objects sets able to explore the best resources of each system, to be combined in unified products, to be freely delivered through the Web thus enabling larger integration and collaboration among developers.

SUMÁRIO

ÍNDICE DE FIGURAS ..... **III**

1. INTRODUÇÃO **1**

1.1. Contexto **1**

1.2. Objetivo **2**
1.2.1. Objetivo Geral 2
1.2.2. Objetivos Específicos 2

1.3. Organização **3**

2. OBJETOS E COMPONENTES DE SOFTWARE EDUCACIONAIS **5**

2.1. Perspectivas da elaboração e do uso de software na educação **5**
2.1.1. Elaboração de software educacional – Professor como Autor 5
2.1.2. Pacotes de software 10
2.1.3. Utilizando a programação de computadores no apoio ao aprendizado 11

2.2. Construindo software educacional com objetos e componentes **15**
2.2.1. Interfaces, linguagens e computadores 15
2.2.2. Porque utilizar objetos e componentes de software? 17
2.2.3. Utilizando objetos e componentes em projetos educacionais 22

2.3. Objetos de conhecimento e metadados **23**

2.4. Objetos e componentes no sistema Casa Mágica **27**
2.4.1. A produção como objeto de estudo 30

3. MODELOS DE OBJETOS PARA WEB **34**

3.1. Fundamentos de um Modelo de Objetos para Web **34**
3.1.1. Representação de um objeto 37

3.2. Representação de Objetos através de Marcação Estruturada **39**
3.2.1. Análise comparativa 55

4. ANIMA **60**

4.1. Estrutura do Projeto **61**

4.2. Modelo **65**
4.2.1. Representação da Composição 65
4.2.2. Representação de Classe, Interface e Metadados (CIM) 74
4.2.3. Mecanismos de conversão 86
4.2.4. Protocolo padronizado de comunicação 88

4.3. Representação baseada em marcação estruturada **96**

4.3.1. Representação da Composição em XML 96
4.3.2. Representação de Classe, Interface e Metadados (CIM) em RDF 98
4.3.3. Mecanismo de conversão utilizando XSLT 102
4.3.4. Protocolo de comunicação entre objetos – Mensagens em XML 105
4.3.5. Aplicando a representação baseada em Marcação Estruturada 107
4.3.6. Utilizando *Anima* 108

4.4. Ferramentas 111
4.4.1. Bibliotecas 112
4.4.2. Utilitários CIM 112
4.4.3. Autoria e execução de composições 113

5. PROJETOS EDUCACIONAIS COM ANIMA 115

5.1. Máquinas e componentes de software na aprendizagem matemática 116

5.2. Casa Mágica e Anima 126

6. CONCLUSÃO 129

7. REFERÊNCIAS BIBLIOGRÁFICAS 138

A. TÓPICOS DE REVISÃO BIBLIOGRÁFICA 147

A.1. XML 147

A.2. RDF 147

A.3. XSLT 148

B. ESQUEMAS 149

C. ESQUEMAS EM XML 150

D. FERRAMENTAS EM JAVA PARA SUPORTE AO PROJETO ANIMA 151

D.1. Bibliotecas 151

D.2. Utilitários CIM 151

E. ESTUDOS DE CASO E APLICAÇÕES PRÁTICAS 152

E.1. Pacotes de Software Educacional – Estudo de Caso 152

E.2. Construção de software utilizando objetos e componentes – Estudo de Caso 152

E.3. Projetos educacionais baseados em objetos e componentes 153

E.4. Integrando e Colaborando 153

ÍNDICE DE FIGURAS

Figura 2–1: Procedimento para a construção de uma caixa em Logo (esquerda) e Boxer (direita) [DIS97] 13
Figura 2–2: Ficha com metadados padrão IMS, associada a um objeto no repositório EOE..... 26
Figura 2–3: Diagrama de funcionamento de uma aplicação produzida em Casa Mágica..... 27
Figura 2–4: Conjunto de Locais de uma aplicação interligados por portas..... 28
Figura 2–5: Local de lançamento de um corpo (projeto de física) e barra mostrando seus objetos..... 29
Figura 2–6: Barra de componentes do sistema Casa Mágica. 29
Figura 2–7: Componente botão é configurado para enviar mensagem para componente relógio..... 30
Figura 2–8: Visores alternativos para o mesmo componente – corpo que descreve trajetória de lançamento oblíquo. 31
Figura 2–9: Interação com o componente "Forma" através do painel de laboratório. 32
Figura 3–1: Diagrama ilustrando a inserção do DomainModel e do DomainObjects em uma ObjectBase. 45
Figura 3–2: Diagrama geral de construção e decodificação de um DomainObjects 47
Figura 3–3: Diagrama de execução de uma aplicação através do BML *player*. 49
Figura 3–4: Diagrama de execução de uma aplicação através do BML *compiler*. 49
Figura 3–5: Diagrama de transformação da DTD em classes Java. 52
Figura 4–1: Deslocamento de representações para uma camada neutra. 62
Figura 4–2: Diagrama geral do projeto *Anima*. 63
Figura 4–3: Diagrama de associação de código a objetos, a partir de um evento, em Java. 66
Figura 4–4: Diagrama de associação de objetos, a partir de um evento, no BeanBox. 67
Figura 4–5: Divisão de espaços de trabalho (*workspaces*) entre objetos escritos em linguagens diferentes. 70
Figura 4–6: Representação de objetos e sua interligação em um espaço de trabalho. 71
Figura 4–7: Dois objetos da mesma classe, com identidade e valor dos atributos diferentes. 72
Figura 4–8: Diagrama do processo de envio de mensagem, baseado na ocorrência de um evento. 73
Figura 4–9: Mapeamento das áreas de trabalho em áreas utilizadas pelos programas. 77
Figura 4–10: Importação da estrutura de uma classe utilizando o mecanismo de reflexão. 79
Figura 4–11: Modelagem e edição de propriedades da classe e interface. 80
Figura 4–12: Gravação e leitura da representação CIM em formato aberto..... 81
Figura 4–13: Processo de importação realizado de forma independente da edição. 82
Figura 4–14: Leitura de representação CIM por uma ferramenta de montagem e edição de composição. 82
Figura 4–15: Instanciação de componente de classe compatível com ambiente de produção. 83
Figura 4–16: Gravação e leitura da composição em arquivo. 84
Figura 4–17: Diagrama de transformação das representações genéricas em uma aplicação implementada em linguagem específica. 86
Figura 4–18: Aplicação de plano de transformação na representação de um objeto..... 88
Figura 4–19: Diagrama de comunicação entre objetos através de *Anima*. 90
Figura 4–20: Modalidades de construção de interface entre objeto e *Anima*. 91
Figura 4–21: *Common Object Request Broker Architecture* [VIN97] 93
Figura 4–22: Comunicação entre Mediadores *Anima* através de CORBA, SOAP, etc..... 95
Figura 4–23: Estado de um objeto representado em XML..... 97
Figura 4–24: Ligação entre dois objetos em um espaço de trabalho. 98
Figura 4–25: Associação dos elementos do Modelo Genérico à tecnologia XSLT. 104
Figura 4–26: Diagrama ilustrando o processo de transformação, que parte da representação XML até o documento apresentado do navegador *Web* 108
Figura 4–27: Relação entre utilitários e Representação CIM em RDF. 112
Figura 5–1: Representação de uma máquina de soma..... 116
Figura 5–2: Programa que utiliza a metáfora das máquinas para exercícios de soma e subtração. ... 117

Figura 5–3: Ilustração do comportamento de uma função através de uma máquina..... 117
Figura 5–4: Máquina-função ilustrada com animais. 118
Figura 5–5: Máquina função composta por máquinas menores. 119
Figura 5–6: Composição em Casa Mágica que apresenta a função y=2x+3. 122
Figura 5–7: Outra composição em Casa Mágica que apresenta a função y=2x+3. 124

1. INTRODUÇÃO

1.1. Contexto

O tema abordado neste trabalho combina o paradigma da orientação a objetos com modernas técnicas de desenvolvimento de aplicações, utilizando componentes de *software*.

Os avanços das tecnologias de objetos e componentes de *software* têm repercutido no contexto educacional. Um número crescente de projetos explora diversos benefícios dessas duas tecnologias. Entre os benefícios, destacamos dois de especial interesse:

- O primeiro resulta de uma nova estratégia para elaboração de aplicações, que se contrapõe à estrutura tradicional dos sistemas de *software* educacional, pautada em grandes programas fechados e monolíticos. Esta nova estratégia se baseia na integração de esforços de uma teia de produtores dos componentes de *software*, os quais são facilmente distribuídos, configurados e combinados na construção de aplicações maiores.
- O segundo está associado às metáforas dos objetos e componentes de *software*, capazes de tornar acessível a construção de aplicações computacionais a professores – autores de produtos para uso em sua prática docente – e alunos que, através de projetos envolvendo a programação de computadores, tornam-se agentes ativos na construção do próprio conhecimento.

Um outro fenômeno é conseqüência do casamento da tecnologia de objetos com a *Web*, resultando no que Frank Manola denomina *Web Object Model* [MAN99]. Os objetos constituem uma tecnologia ideal para a *Web*, pois entre outras coisas:

- São capazes de esconder detalhes de implementação de diferentes plataformas, comunicando-se através de interfaces padronizadas. Isto é bastante adequado para a heterogeneidade da Internet.
- Combinados com linguagens de código móvel, os objetos podem compor unidades altamente autônomas e portáteis.

Diversas propostas de modelos de objeto para *Web* (*Web Object Model*) têm utilizado a linguagem XML [BRA00] como base para representação de seus dados, não apenas pela grande aceitação desta linguagem na *Web*, como também por ser XML uma metalinguagem

dotada de recursos para a definição de linguagens especializadas, conforme a necessidade de cada comunidade.

A soma das contribuições proporcionadas por cada uma destas tecnologias – objetos, componentes, linguagens de código móvel e XML – forneceu subsídios para a construção do projeto *Anima*, que é o principal tema tratado nesta dissertação.

Objetos e componentes são a base do modelo de *Anima*. XML define um substrato sobre o qual é montado o conjunto de representações do projeto, conferindo-lhe características de neutralidade – no que diz respeito à plataforma e implementação – e tornando-o adequado para a *Web*.

Este conjunto de características permitiu a construção de um modelo aberto e independente de plataforma, utilizado para a construção de *software* educacional, que elege a *Web* como foro para o desenvolvimento de projetos amplamente participativos e colaborativos.

1.2. Objetivo

1.2.1. Objetivo Geral

O objetivo geral da dissertação é propor um modelo genérico, independente de plataforma e de linguagem de programação, bem como sua versão em uma linguagem XML, adequada à representação de classes, de objetos de *software* e ao desenvolvimento de aplicações baseadas nos mesmos. Adicionalmente, o trabalho contribui com ferramentas que exploram a linguagem na construção de aplicações educacionais capazes de trafegar pela *Web*.

1.2.2. Objetivos Específicos

- Propor um modelo genérico adequado à representação de classes, de objetos de *software* e ao desenvolvimento de aplicações baseadas nos mesmos, cuja estrutura seja independente de plataforma e de linguagem de programação.
- Propor uma tradução do modelo em linguagem XML, de modo que as produções possam ser distribuídas através da *Web*.
- Definir mecanismos de conversão padronizados, que realizem a transposição dos objetos e das aplicações representados através do modelo genérico para implementações específicas.

- Implementar ferramentas que permitam explorar o modelo proposto na elaboração e execução de uma classe de aplicações voltadas à área educacional.

1.3. Organização

A dissertação está estruturada em cinco capítulos. Os dois primeiros apresentam os fundamentos teóricos que nortearam a elaboração deste trabalho e os dois subseqüentes apresentam o projeto *Anima*. O último apresenta uma reflexão dos resultados do projeto e propostas para projetos futuros.

Por ser um projeto que envolve a aplicação de tecnologia da informação no domínio da educação, o primeiro capítulo – **Objetos e Componentes de *Software* Educacionais** – preocupa-se em situar o projeto no contexto da educação. São abordadas questões relacionadas ao desenvolvimento e uso do *software* educacional sob diversas perspectivas, descrevendo-se algumas tecnologias usadas para este fim e suas limitações. Introduz-se a tecnologia de objetos e componentes como um avanço qualitativo na produção desse tipo de *software*, no reaproveitamento e desenvolvimento de trabalhos colaborativos. No final do capítulo são apresentados projetos que fazem uso de objetos e componentes de *software* na educação, destacando-se algumas de suas limitações.

No capítulo seguinte – **Modelos de Objetos para *Web*** – são introduzidas algumas das tecnologias de informática sobre as quais é montado o projeto. Apresenta-se um resumo de modelos encontrados na literatura, propostos para a integração entre o paradigma da orientação a objetos e a *Web*. Em especial, utiliza-se a abordagem de Frank Manola referente ao *Web Object Model* [MAN99].

O capítulo ***Anima*** descreve o projeto. Ele está dividido em três seções que partem de uma abordagem mais genérica, em direção a uma mais específica. Inicialmente é traçado o modelo abstrato sobre o qual está estruturado todo o projeto. A partir deste modelo pode-se alcançar um leque amplo de possíveis implementações. Uma delas é apresentada na segunda parte, fazendo uso de XML e seus derivados. A terceira parte descreve a implementação de um conjunto de ferramentas baseadas no modelo apresentado e sua respectiva tradução em XML. Estas ferramentas são um referencial de implementação do modelo e servem para verificar sua eficácia.

O capítulo **Projetos Educacionais com *Anima*** apresenta um estudo de caso envolvendo as ferramentas produzidas. Este estudo de caso faz uso dos recursos mais significativos de *Anima* e ilustra seu potencial na aplicação da informática na educação. O

texto é concluído com o capítulo **Considerações Finais**, no qual o resultado do projeto é avaliado. São discutidas as metas alcançadas e as limitações encontradas no modelo proposto. Algumas destas limitações e novas possibilidades encontradas durante a elaboração do projeto deram origem a propostas de futuros projetos, apresentadas também nesse capítulo.

2. OBJETOS E COMPONENTES DE *SOFTWARE* EDUCACIONAIS

Este capítulo apresenta um panorama geral do uso do *software* educacional em atividades de ensino-aprendizagem, definindo algumas limitações inerentes aos atuais modelos adotados para a construção deste tipo de *software*. Como resultado deste capítulo, obtemos um quadro do contexto dentro do qual se insere o projeto *Anima*.

2.1. Perspectivas da elaboração e do uso de *software* na educação

Este tópico desenvolve uma reflexão sobre a seguinte questão: "Quais as limitações encontradas na atual metodologia de desenvolvimento de *software* na educação, que nos levam a crer na necessidade de uma nova abordagem?".

Para isto, traçamos um panorama do modo como tem sido desenvolvido e utilizado o *software* em atividades de ensino-aprendizagem, de modo especial por instituições de ensino, e destacamos diversas limitações encontradas neste processo, principalmente em conseqüência de sua estrutura de construção tipicamente monolítica.

O texto enfoca três perspectivas, conforme a postura do usuário no que diz respeito ao uso ou à construção de um *software*: a do professor autor de *software*, a do uso de pacotes de *software* prontos ou a do aluno que utiliza a atividade de programação como mediadora do seu aprendizado.

As limitações destacadas nesta análise fornecem subsídios para delinearmos a problemática, sobre a qual lançamos a proposta deste projeto.

2.1.1. Elaboração de *software* educacional – Professor como Autor

Quando os primeiros microcomputadores começaram a ser utilizados no ensino, o *software* disponível era bastante limitado. Havia alguns programas do tipo: 'estudo dirigido', implementações da linguagem Logo [PAP85], aplicativos genéricos que podiam ser utilizados em atividades educacionais (como processadores de texto) e programas mais simples para disciplinas específicas.

Motivados pelo potencial de aplicação dos computadores na educação, alguns professores idealizavam projetos associados às suas disciplinas, para os quais não havia qualquer *software* que pudesse atender a sua demanda. Uma parte destes professores se envolvia diretamente na construção de programas de computador para atender à própria

necessidade e, para isto, tinham à disposição linguagens de programação genéricas, que não eram projetadas especificamente para uso educacional.

Os professores produziam programas dentro de suas limitações de tempo e conhecimentos de programação. Este tipo de programa, em geral, atendia a necessidades próprias, era tipicamente pequeno, bastante específico e construído conforme a organização e metodologia que o autor-professor adotava na disciplina onde o utilizava. Não possuía flexibilidade para se adaptar a grandes variações existentes entre professores, escolas, regiões e países diferentes. Por isto, dificilmente ele era utilizado em outras instituições de ensino e, em grande parte dos casos, nem era utilizado nem mesmo por outros professores da disciplina na mesma escola.

Somado a isso, o uso de linguagens e sistemas baseados em uma plataforma específica dificulta o intercâmbio de programas. Em algumas situações, os sistemas exigem a instalação de bibliotecas e *drivers* juntamente com o programa, tornando bastante trabalhosa a tarefa de transferência do mesmo para outros computadores.

A crescente quantidade de escolas que passaram a utilizar computadores em suas atividades de ensino-aprendizagem estimulou empresas, instituições e autores de *software* a se empenharem no desenvolvimento de produtos mais abrangentes e flexíveis.

Apesar do surgimento de produtos sofisticados de *software* educacional, os professores, em sua maioria, ainda aproveitam pouco destes sistemas, seja porque esses produtos não estruturam o conteúdo da disciplina como os professores desejam, porque não contemplam alguns tópicos lecionados, ou porque possuem uma estrutura e conjunto de recursos que não satisfazem suas expectativas. Em alguns aspectos, estes sistemas ainda não se mostram suficientemente abertos e flexíveis para o nível de adaptação solicitado. Como constata Jeremy Roschelle: “Aplicações educacionais têm que ser muito flexíveis porque o currículo e estilos de ensino variam tremendamente entre instituições, locais, a até mesmo entre instrutores da mesma instituição” [ROS99].

Andrea diSessa ressalta a questão da diversidade em confronto com a crescente necessidade de tornar o *software* adaptativo:

> Educação é fundamentalmente uma prática local e culturalmente específica. Há uma grande diversidade nas práticas dos professores, uma diversidade que a pesquisa mostrou nos remeter a uma diversidade no conhecimento dos professores sobre o assunto, seu repertório de estratégias para ensinar conceitos particulares (freqüentemente referenciado como conhecimento de conteúdo pedagógico), suas crenças sobre sua função como professores, suas crenças sobre aprendizes e aprendizado, e suas crenças sobre a natureza do conhecimento e o que eles acreditam que deve ser

> ensinado (Fennoma & Franke, 1986; Shulman, 1986; Thompson, 1992). A menos que a tecnologia educacional seja adaptativa, os professores terão dificuldades em modificar seu estilo de trabalho, fruto de longa experiência, para ajustá-lo ao computador, ao invés do contrário. [DIS01]

Não encontrando soluções que lhes pareçam adequadas, ou sistemas flexíveis o bastante para atender a suas necessidades, os professores continuam interessados no desenvolvimento de aplicações específicas.

As ferramentas de desenvolvimento evoluíram, adequando-se e tornando-se mais produtivas para a criação de *software* educacional. Os sistemas de autoria apresentam interfaces mais intuitivas, utilizando metáforas como: ícones, páginas, linha de tempo, etc.

No ano de 1998 desenvolvemos um projeto no Colégio Anchieta[1], com uma ferramenta de autoria denominada Authorware[2]. Os professores foram treinados para utilizá-la na construção de aplicações destinadas à sala de aula. Durante alguns anos, acompanhando o desenvolvimento de aplicações por professores, observamos o seguinte:

- As ferramentas ainda exigem um grande esforço para o desenvolvimento de pequenas aplicações. As bibliotecas de recursos não são especializadas, o que exige que a maioria dos professores inicie seu trabalho praticamente da estaca zero.
- Aplicações simples envolvendo apresentação de recursos multimídia, que constituem o alvo principal das ferramentas de autoria, são facilmente produzidas por professores. Por outro lado, quando estão envolvidos elementos mais complexos de simulação e interatividade, estes sistemas exigem um aprofundamento em recursos de programação, que foge ao alcance da maioria dos professores.

A nossa interação com escolas de ensino fundamental e ensino médio que utilizavam outras ferramentas de autoria, somada a testes e projetos-piloto desenvolvidos com diversas destas ferramentas, demonstraram-nos que as observações apresentadas se aplicam igualmente a outras ferramentas também populares no mercado.

A questão parece conduzir a um dilema, resumido em duas questões:

---

[1] O Colégio Anchieta é uma instituição que atua no ensino fundamental e médio em Salvador – Bahia, que se tem destacado por oferecer um ensino de qualidade. Nesta instituição, o autor deste projeto tem atuado como coordenador de informática aplicada à educação desde 1993. Deste modo, muitos projetos e reflexões, relatados neste trabalho, foram realizados com professores e alunos do colégio.

- Será que os sistemas atuais de *software* educacional se tornarão suficientemente flexíveis e adaptativos, a ponto de dispensarem a construção de aplicações computacionais específicas pelos professores?
- Ou será que, por mais que estes sistemas evoluam, sempre haverá a necessidade de se criar programas e, portanto, as ferramentas de produção precisam evoluir para tornar cada vez mais viável ao professor a criação de *software*?

Este dilema deriva do modo como está estruturada a maioria dos sistemas de *software* educacional prontos para uso (pacotes de *software*): blocos monolíticos fechados. Conseqüentemente, se o professor quer realizar alguma adaptação ou modificação não prevista pelo sistema original, ele se depara freqüentemente com uma grande caixa preta, contendo código de computador, que dificilmente pode ser modificada.

Diante desta situação, muitos optam pelo desenvolvimento de um novo sistema partindo do zero, sem que consigam aproveitar qualquer parte do sistema de que já dispõem.

Programação é para programadores?

Desde o início desta discussão, colocamos o professor como figura central no desenvolvimento do *software*. Contudo, não há dúvidas que, embora a programação de computadores seja uma tarefa típica dos profissionais de informática, há muito tempo deixou de ser uma tarefa exclusiva dos mesmos. A simplificação e popularização de ambientes de programação têm diminuído a distância entre programas de computador e usuários leigos, muitos dos quais tem se aventurado no desenvolvimento de programas.

As instituições de ensino, onde está concentrada a maior demanda de *software* educacional, em geral não dispõem de uma equipe de informática para este tipo de necessidade. O número de profissionais necessários para atender à demanda, ainda que parcialmente, é muito grande, tornando seu custo proibitivo.

Em uma experiência que realizamos no Colégio Anchieta, foi empregada uma equipe de três profissionais de informática no desenvolvimento de *software* educacional, para atender às necessidades apresentadas pelos professores. A produção resultante se mostrou bastante eficaz, sendo utilizada e solicitada pelos professores por anos consecutivos. Entretanto, não foi possível atender mais que 10% da demanda.

---

[2] **Authorware** – sistema de autoria produzido pela Macromedia, Inc. A página do Authorware pode ser encontrada no endereço http://www.macromedia.com/software/authorware/.

Ampliar a equipe de programadores envolvida no desenvolvimento do *software* educacional se mostra como uma possível solução. No entanto, o atual panorama da informática tem apontado para outra direção.

A melhoria da interface de ferramentas destinadas ao desenvolvimento de aplicações educacionais, tais como as ferramentas de autoria, em conjunto com a evolução de seus recursos têm conduzido a uma nova postura, na qual o professor se torna autor das próprias aplicações, tal como já o faz com textos, slides, etc.

Se o professor não é um especialista em programação de computadores, ele é, por outro lado, um especialista em educação. Ele conhece muito bem o contexto no qual pretende utilizar o *software* a ser produzido.

Ao projetar o Smalltalk [ING81] como uma linguagem onde usuários leigos desenvolvem suas próprias aplicações para necessidades individuais, a equipe do projeto partiu do princípio: “Se um sistema vai servir ao espírito criativo, ele tem que ser inteiramente compreensível para o indivíduo isolado”. [ING81]

Este dilema de quem desenvolve o *software* também está ligado ao modo como são construídas as aplicações atualmente. Duas características desejáveis nas ferramentas de programação ainda carecem de aperfeiçoamento:

- A simplicidade na elaboração de um produto onde colaborem educadores e programadores, de tal modo que cada um se concentre na sua especialidade.
- O fácil reaproveitamento de produtos desenvolvidos por outras equipes.

A primeira característica permite que o profissional de informática se especialize na construção dos instrumentos utilizados pelos educadores na elaboração do produto final. Este último fica livre para desenvolver seu modelo diretamente no computador. Desta forma o único mediador entre a sua idéia e o produto final é o sistema computacional que ele usa. Daniel Ingalls resume a questão:

> O ponto aqui é que o potencial humano se manifesta em indivíduos. Para realizar este potencial, nós precisamos prover um meio que possa ser controlado por um único indivíduo. Qualquer barreira que exista entre o usuário e alguma parte do sistema será eventualmente uma barreira para a expressão criativa. [ING81]

2.1.2. Pacotes de *software*

No tópico anterior foram citadas algumas questões referentes à estrutura e às limitações dos sistemas de *software* educacional predominantes, produzidos por empresas, instituições ou autores individuais e comercializados ou distribuídos gratuitamente. Denominamos estes sistemas, genericamente, de pacotes de *software*.

Com o intuito de introduzir as questões que envolvem o uso de pacotes de *software* em atividades de ensino-aprendizagem, optamos por apresentar um estudo de caso que confronta três pacotes em uma atividade planejada por um professor, para ser utilizada em sala de aula.

O tema escolhido para estudo de caso envolve principalmente modelos matemáticos interativos e a simulação. Utilizar simulações com os alunos propicia ótimas condições para que estes atuem ativamente no processo de aprendizagem, construindo e experimentando modelos, analisando resultados, modificando parâmetros, etc.. Por tal motivo, este tema tem despertado a atenção de pesquisadores e educadores: "Em um relatório sobre tecnologia educacional para o Presidente dos Estados Unidos, um comitê de consultores científicos apresentaram as mais promissoras aplicações construtivistas da tecnologia, com simulações no topo da lista". [REP99]

Este tipo de prática, pautada sobre a ótica do Construcionismo [PAP94], constitui o domínio onde fundamentaremos nossa análise e proposta de atuação.

O Construcionismo fundamenta-se numa perspectiva onde o aprendizado é encarado como uma atitude ativa, onde o aluno constrói o próprio conhecimento. No uso dos computadores, sob esta ótica o aluno, através de um software apropriado, aprende exercitando a tarefa de "ensinar" o computador [VAL93].

A opção pela matemática se deu pela riqueza de bons produtos disponíveis nesta área, dotados de grande flexibilidade. Além disto, há alguns anos, o autor deste trabalho vem desenvolvendo projetos com este tipo de *software*, em conjunto com professores de matemática, o que se traduziu em um rico conjunto de experiências quanto a vantagens e limitações encontradas em cada produto.

<u>Estudo de Caso</u>

No Apêndice E.1, apresentamos o estudo de caso realizado com três softwares educacionais:

- **Winplot** – plotagem de curvas e superfícies;
- **Modellus** – criação, simulação e análise de modelos matemáticos;
- **Homos** – simulação baseada em autômatos celulares.

Sintetizamos a seguir as constatações mais relevantes que servirão para o nosso estudo:

Como acontece com freqüência entre pacotes de *software* deste tipo, apesar de possuírem muitos recursos semelhantes, tais como traçadores de gráficos cartesianos, sistemas de interpretação de modelos matemáticos, etc., cada um deles desenvolve seus recursos de forma independente, sem aproveitar o que já foi construído em outras aplicações.

A integração, em geral, não existe ou é muito pequena. Fica limitada a recursos de cópia e colagem de trechos de texto, números ou gráficos. Sua estrutura monolítica impede a combinação e aproveitamento de componentes individuais de *software* do conjunto.

Alguns professores gostariam de inserir animações produzidas por um dos pacotes de *software* analisados, na forma de objetos dentro de seus diapositivos, preparados para apresentação em sala de aula através de um *software* específico.

Professores, usuários de uma ferramenta de autoria, estariam interessados em desenvolver projetos que aproveitassem recursos destes pacotes em suas aplicações. Recursos de animação baseados em modelos matemáticos e simulação do comportamento de um grande volume de objetos autônomos não estão disponíveis na ferramenta de autoria que eles utilizam (nem disponíveis nas principais ferramentas de autoria).

Tais demandas não encontraram uma solução adequada nos atuais pacotes pois, como já foi enfatizado, estes sistemas são construídos sobre uma estrutura monolítica e não são passíveis das modalidades de integração apresentadas.

Por outro lado, utilizando a tecnologia dos objetos e principalmente dos componentes de *software*, os sistemas são capazes de se integrar de uma forma bastante natural, além disto, partes de sua estrutura podem ser reaproveitadas em outras produções.Isto será apresentado e demonstrado posteriormente.

2.1.3. Utilizando a programação de computadores no apoio ao aprendizado

Nos meados da década de 70 Seymour Papert liderou o desenvolvimento de Logo, que envolve ao mesmo tempo uma linguagem de programação, projetada para ser utilizada em

atividades de ensino-aprendizagem, e uma filosofia de trabalho baseada no Construcionismo. Logo representou um marco importante no uso de computadores na educação, pois foi a primeira proposta, com ampla difusão, onde o aluno assumia um papel ativo na construção de seu conhecimento.

Por ser uma linguagem bastante aberta, Logo teve como principal alvo não uma área de conhecimento específico, apesar de poder ser direcionado para tanto, mas sim a própria estratégia como o aprendiz organiza, estrutura e conduz seu aprendizado: "... e ao ensinar o computador a ‘pensar’, a criança embarca numa exploração sobre a maneira como ela própria pensa. Pensar sobre modos de pensar faz a criança tornar-se um epistemólogo, uma experiência que poucos adultos tiveram." [PAP85]

A concepção da criança como um “epistemólogo” influenciou diversas propostas subseqüentes de aplicação do computador na educação e faz parte do contexto de uma discussão mais ampla sobre o próprio aprendizado.

Ao propor o uso de uma linguagem de programação no contexto educacional, Papert pretendia não restringir a exploração do recurso computacional aos limites estabelecidos por parâmetros de configuração, típicos de muitos sistemas que possibilitam a intervenção do usuário. Por outro lado, a generalidade destas linguagens pode ser um problema quando se pretende explorar um domínio específico. O aprendiz-programador é obrigado a seguir um caminho muito longo de modelagem para alcançar sua meta.

Com a popularização de microcomputadores dotados de recursos multimídia mais sofisticados, tais como sistemas de vídeo de alta resolução, placas de som, processadores velozes, etc., apareceram e se difundiram rapidamente as ferramentas de autoria. Conforme já foi apresentado, estas ferramentas podem tanto ser utilizadas por professores na construção de aulas ou aplicações a serem utilizadas com seus alunos, como também podem ser utilizadas diretamente com os alunos, que se tornam os autores.

Este tipo de ferramenta é adequado ao desenvolvimento de muitas atividades de ensino-aprendizagem. Com ela os alunos facilmente integram diversas mídias, elaboram animações simples e implementam rotinas básicas de interação. Por outro lado, como já foi observado com os professores, dado seu domínio específico de atuação, quando um aluno necessita de um recurso mais avançado, não disponível no arsenal básico da ferramenta, ele deve fazer uso de recursos de programação que exigem um conhecimento mais aprofundado da ferramenta, o que é, em muitos casos, pedagogicamente inadequado.

Refletindo sobre as mudanças e benefícios advindos dos recursos de apresentação e interação, oferecidos pelas novas interfaces gráficas dos sistemas operacionais, Andrea diSessa projetou o Boxer [DIS97] como um candidato à sucessão do Logo. Ele mantém a concepção baseada em uma linguagem de programação aberta: “Ambos os projetos acreditam que a programação de alguma forma, é essencial para liberar verdadeiramente o poder do computador como uma ferramenta de aprendizagem” [DIS97], modificando o modo como os programas são estruturados e construídos.

Para diSessa o projeto Logo teve sua concepção original baseada nos antigos teletipos, nos quais a interação era essencialmente textual. Por este motivo, a principal forma de atuação em qualquer sistema Logo é baseada em instruções descritas textualmente. Isto se manteve até mesmo nas mais novas versões desta linguagem.

O Boxer, por sua vez, utiliza a metáfora de caixas (Figura 2-1), que contêm os mais diversos elementos de um sistema, como: comandos, variáveis, objetos, etc., para tornar os comandos abstratos do Logo elementos concretos e visíveis, que podem ser modificados e experimentados por manipulação direta. Esta concepção resgata a metáfora dos componentes de *software*.

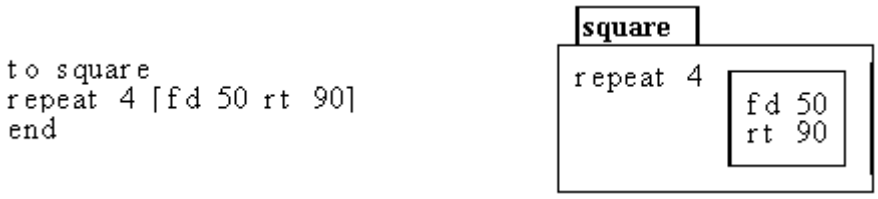


Figura 2-1: Procedimento para a construção de uma caixa em Logo (esquerda) e Boxer (direita) [DIS97]

Sistemas baseados em componentes conseguem aliar a facilidade de criação e experimentação das ferramentas de autoria a uma capacidade extra de permitir que o autor desenvolva novos componentes ou utilize componentes criados por outros autores, conforme sua necessidade.

O Boxer desenvolve uma proposta onde o próprio autor da aplicação, tal como o professor ou aluno, é o criador de seus novos componentes, que são posteriormente compartilhados com os demais. Conforme relata Andrea diSessa, sua estrutura organizada em caixas, que podem ser testadas, analisadas e modificadas em um ambiente visual e interativo, estimula a colaboração entre alunos: “Quando um programa é fácil de inspecionar, remanejar e modificar, ele é muito mais fácil de compartilhar” [DIS97].

Nos últimos anos a popularização da *Web* traz à tona, de uma forma nova e bastante potencializada, a hipermídia, traçando uma grande rede de hiper-ligações por todo o mundo. Páginas HTML passaram a ser produzidas por uma imensa quantidade de pessoas e instituições. Produzir páginas e publicá-las na Internet, tem se mostrado uma atividade pedagógica de grande riqueza.

O processo de produção das páginas desenvolve a capacidade de reunir, sintetizar e organizar informações que serão dispostas nas mesmas, bem como habilidades artísticas para a apresentação estética dessas informações, em conjunto com cores, imagens e outras mídias disponíveis. Neste contexto, o aluno ultrapassa o papel de mero receptor de informações e passa a ser atuante.

Se por um lado a simplicidade da linguagem HTML foi um dos grandes responsáveis pela rápida e ampla adesão de diversos autores à *Web*, por outro, esta linguagem atua em um domínio bastante específico, que engloba a organização e apresentação de informações nas suas diversas formas (mídias). Isto significa que esta linguagem não dispõe de recursos mais abrangentes, oferecidos pelas linguagens de programação.

Em 1995, a linguagem de programação Java se apresentou como uma solução para o desenvolvimento de programas na *Web*. Alguns aspectos de sua concepção, como a independência de plataforma e a capacidade de embutir pequenos programas em páginas HTML, entre outros, tornaram Java uma linguagem bastante adequada para a *Web*.

Enquanto a simplicidade de HTML tem contribuído para seu uso em atividades de ensino-aprendizagem, quando tais atividades envolvem programação de computadores, o uso de Java, como o de outras linguagens de programação para *Web*, mostra-se muito complexo.

Concluímos que, embora a programação de computadores se apresente como um precioso recurso pedagógico, diversas características são desejáveis, tais como: a simplicidade na composição, a inspeção e a edição dos programas (especialmente aquela propiciada por ambientes visuais), uma estrutura que estimule o intercâmbio e o trabalho colaborativo e a capacidade de se integrar de forma natural com a *Web*. Modelos baseados em objetos e componentes têm o potencial de combinar tais características, conforme apresentamos a seguir.

2.2. Construindo *software* educacional com objetos e componentes

No final da década de 60, surgiu uma linguagem de programação de computadores que se destacou por adotar a metáfora de objetos intercomunicantes para representar os módulos de um programa. Ela daria origem ao que foi chamado, posteriormente, de Programação Orientada a Objetos, que seria o ponto de partida para um paradigma que se expandiu para outras áreas da computação, tais como a análise e o projeto de sistemas, os bancos de dados, etc. Nos últimos anos, modelos computacionais baseados em objetos passaram a ser utilizados em outros contextos, inclusive no da educação.

Curiosamente, apesar de seu grande sucesso ter se dado originalmente no domínio dos iniciados em computação, o principal propósito de um dos primeiros projetos – que deu verdadeiro impulso ao paradigma da orientação a objetos – era tornar a construção de aplicações em computadores acessível a qualquer usuário leigo.

Em paralelo com a metáfora dos objetos, estruturou-se a dos componentes de *software*. Estes últimos dão grande ênfase ao planejamento e à construção de modelos computacionais baseados em blocos de código (componentes) autônomos, facilmente configuráveis e combináveis entre si. Diversas ferramentas baseadas em componentes os estruturam de tal modo, que os autores do modelo não necessitam de conhecimento específico da forma como os blocos foram construídos. Por este motivo, os componentes representaram um grande avanço dos computadores em direção aos autores de projetos, leigos em computação. Como analisaremos adiante, mesmo se os componentes não são necessariamente associados a objetos, a integração entre objetos e componentes potencializa a atuação de ambos.

2.2.1. Interfaces, linguagens e computadores

Cada instrução escrita em uma linguagem computacional ou comando dado a um sistema operacional é passível de um mapeamento para uma estrutura interna de representação do computador. Por este motivo, o usuário, ao interagir com o computador através destes meios, está lidando com uma metáfora que lhe é apresentada com o objetivo de simplificar a comunicação ou aumentar sua produtividade.

As metáforas escolhidas nas primeiras linguagens e interfaces, partindo do computador e sua estrutura, são adequadas quando o indivíduo que interage com a máquina é um profissional da área de informática. Por outro lado, quando se trata de usuários leigos, isto se torna um entrave.

Interface e Novas Metáforas

Lakoff e Johnson ampliam a interpretação do termo metáfora para além dos limites da linguagem, analisando sua influência no pensamento e na ação do indivíduo.

> A metáfora é, para muita gente, um dispositivo da imaginação poética e da alegoria retórica – um assunto de linguagem extraordinária, ao invés de ordinária. Além disto, metáfora é tipicamente vista como característica só da linguagem, uma questão de palavras, ao invés de pensamento e ação. Por esta razão, muitas pessoas pensam que podem avançar perfeitamente bem sem metáforas. Nós percebemos, ao contrário, que essa metáfora impregna cada dia da vida, não somente na linguagem, mas no pensamento e ação. Nosso sistema conceitual normal, em termos do que pensamos e agimos, é fundamentalmente metafórico por natureza. [LAK80]

A compreensão da metáfora, inerente às interfaces e linguagens computacionais, passou a ser usada como um instrumento para a sua melhoria, de modo que os ambientes apresentados ao usuário lhes sejam familiares e encontrem-se dentro de seu domínio de conhecimento. Cabe ao computador e a um conjunto de programas apropriados realizar um "mapeamento metafórico", ou seja, converter as ações do usuário, pautadas neste novo modelo, em comandos que deverão estar de acordo com o funcionamento específico da máquina.

As atuais interfaces de Sistemas Operacionais são um exemplo típico. Com o surgimento da interface gráfica, criou-se a metáfora da mesa de trabalho, onde o computador cria um ambiente virtual para o usuário. Os programas são apresentados em janelas, que podem ser deslocadas e reorganizadas pela tela e o acionamento de um programa pode ser feito através de um ícone ou um menu de opções.

Desta maneira, estes sistemas tornaram-se mais fáceis de ser manuseados, mais acessíveis e amigáveis. Isto se deu graças a um deslocamento da perspectiva. A arquitetura e funcionamento do computador deixaram de ser o centro a partir do qual são elaborados os modelos de interface. O centro tornou-se, então, o próprio indivíduo, sua maneira de observar e conceber o mundo que o cerca.

Uma modificação semelhante tem ocorrido com as linguagens de programação modernas.

> As linguagens usadas para programação são definidas, habilitadas e limitadas por seus modelos e metáforas subjacentes. Enquanto a ciência da computação empenha-se por definições formais de sua matéria, a tarefa prática de entender entidades computacionais depende da técnica informal de tomar emprestadas a terminologia e a estrutura de domínios familiares através de mapeamentos metafóricos. [TRA96]

A seguir apresentaremos como a metáfora dos objetos, aplicada ao contexto das linguagens de programação, modificou o modo de se conceber e estruturar programas de computador, tornando o modelo mental e o modelo computacional mais próximos.

2.2.2. Porque utilizar objetos e componentes de *software*?

A diferença entre objetos e componentes é bastante sutil, principalmente por que os componentes cada vez mais se apresentam como uma manifestação especial de objetos. Durante o restante do trabalho, estaremos utilizando ambos os termos em diversos contextos.

Utilizaremos 'objeto' para representar mais genericamente qualquer tipo de objeto de *software*, conforme os princípios da programação orientada a objetos. Este grupo envolve o dos componentes de *software*, mas não se restringe a ele.

Apesar de não ser obrigatória a associação entre componentes e objetos, no restante deste texto tomaremos 'componente' como uma manifestação específica de objeto de *software*, com suas respectivas peculiaridades, como: autonomia, facilidade de configuração, independência do ambiente, etc.. As características e particularidades da associação objeto-componente são discutidas no Apêndice E.2.

Objetos e componentes apresentam-se, aqui, sob duas perspectivas. Por um lado, são metáforas que mediam o modo como pensamos algo e como expressamos este pensamento no computador; por outro, constituem uma forma de estruturar as produções computacionais, de tal maneira que seus elementos de composição de *software* tornam-se fáceis de interligar, trocar e reaproveitar. Ambas as perspectivas têm sido exploradas em projetos educacionais com o uso de computadores.

A seguir, apresentamos algumas das principais vantagens do uso de objetos e componentes de *software* em projetos educacionais. Os itens estão organizados de tal modo que possam ser confrontados com a problemática apresentada no tópico **Perspectivas da elaboração e do uso *de* software na educação**.

Do ponto de vista do autor de *software*:

- **Objetos constituem uma metáfora mais próxima ao modo como percebemos o mundo.**

Conforme já foi abordado anteriormente, a metáfora dos objetos aproxima a maneira de conceber programas de computador ao modo como percebemos o mundo que nos cerca, principalmente quando é confrontada com técnicas tradicionais, como a programação procedural estruturada.

Isto traz benefícios tanto para o professor-autor que pretende construir uma aplicação a ser usada com seus alunos, como também para o aluno-autor, envolvido em algum projeto que utiliza programação de computadores como estratégia pedagógica.

- **Componentes são facilmente configuráveis e podem ser adequados a necessidades particulares.**
- **Os componentes produzidos podem ser combinados, ou recombinados com outros, não limitando sua produção a um programa específico.**

A estrutura de construção dos objetos propicia uma separação eficiente entre sua lógica interna e sua interface com o exterior, o que lhes confere autonomia para serem configurados e combinados externamente. Nos componentes, há uma preocupação extra com esta autonomia e com a simplicidade em sua configuração (como ocorre nas ferramentas visuais), uma vez que são projetados para serem utilizados como peças na construção de diversos tipos de sistema.

Esta separação sugere uma metodologia de trabalho na qual especialistas em informática são os responsáveis pela construção de bibliotecas de componentes. Os autores da aplicação final, geralmente especialistas no domínio da aplicação, como professores, se concentram na configuração e interligação destes componentes. O ESCOT [ROS98], abordado a seguir, desenvolve um projeto conforme este tipo de metodologia.

> Este fato causa um dilema de quem é realmente o público para os componentes de *software* educacionais: São os componentes destinados primariamente aos programadores tradicionais ou deslocam a carga da produção dos programadores para os educadores? Nenhuma resposta é suficiente: Aos programadores faltam horas de contato com estudantes para criar diretamente um excelente *software* educacional, e há poucos indícios de que educadores se tornarão algum dia eficientes criadores de *software* educacional. Então, talvez seja melhor perguntar, ‘Como uma estratégia de componentes tornará o desenvolvimento de *software* e a especialidade em autoria conforme o currículo escolar mais próximos?’ [ROS99]

- **Os fatores anteriores, aliados à facilidade de transportar componentes como pacotes autônomos e integráveis, promovem amplamente o seu intercâmbio entre autores diversos, e a Internet facilita e potencializa este intercâmbio.**

Jeremy Roschelle ressalta a necessidade de uma plataforma comum, padronizada, sobre a qual programadores – principalmente os amadores – possam estabelecer o intercâmbio e integrar suas produções [ROS99], de modo que estas não fiquem limitadas a um uso pessoal ou local, conforme abordado no tópico anterior.

Uma estrutura baseada em objetos e componentes de *software* estabelece condições propícias para a configuração de um padrão utilizado na produção e intercâmbio de peças de *software*, especialmente em um ambiente heterogêneo como a Internet. Por este motivo, como analisaremos mais adiante, têm surgido projetos com esta proposta, especialmente no ambiente educacional.

Além de proporcionar simplicidade, os componentes permitem que o intercâmbio seja feito em escalas variadas. Um autor pode compartilhar uma aplicação completa, ou componentes individuais que serão utilizados em outras produções.

- **Não será necessário partir da estaca zero em cada nova produção, pois os componentes criados podem ser reutilizados, assim como os componentes produzidos por terceiros.**

Bibliotecas de componentes podem ser encaradas como os átomos para a construção de aplicações:

> No futuro, as idéias para atividades educacionais específicas vão converter-se em software tão fácil quanto a escrita de um documento: O processo de desenvolvimento será substituído por um processo de edição, criação e manipulação de ‘aplicações editáveis’. Estas aplicações consistirão em objetos computacionais de alto nível, disponíveis como blocos tangíveis de construção [ROS99].

Estar diante de uma tela em branco, onde toda a aplicação deverá ser forjada, baseada na combinação de um conjunto limitado e simples de primitivas fornecidas por uma ferramenta de autoria, é um dos fatores que mais desestimulam o autor, principalmente leigo, a dar prosseguimento em seu projeto.

Além disto, conforme foi observado no tópico **Elaboração de *software* educacional – Professor como Autor**, o professor tem dificuldade em explorar recursos mais complexos de simulação ou interatividade de uma ferramenta de autoria, pois acaba tendo que se envolver com detalhes de codificação, apropriados apenas para especialistas em informática.

Em ambos os casos, uma estrutura baseada em componentes se apresenta como um caminho satisfatório.

O professor-autor pode enriquecer seu ambiente de produção com bibliotecas de componentes, algumas especializadas no domínio da aplicação que quer desenvolver. Ele alimentará sua biblioteca a partir de diversas fontes: componentes reaproveitados de outra aplicação produzida anteriormente, componentes provenientes de aplicações adquiridas – tais como pacotes de *software* – e, também, componentes provenientes de bibliotecas comercializadas ou distribuídas publicamente por meios de grande difusão, como a Internet.

Quando é necessário um recurso mais complexo e não existem componentes disponíveis para resolvê-lo, é possível solicitá-lo a um programador especializado, conforme a divisão de trabalho sugerida anteriormente. O professor os anexará em sua ferramenta de autoria e os utilizará na elaboração da sua composição, sem se envolver com detalhes de codificação mais especializada.

Do ponto de vista da aplicação:

- **Pacotes de *software* podem se especializar em um domínio, ao mesmo tempo que seus componentes, ou componentes por eles produzidos, combinam-se com os componentes de outros sistemas ou bibliotecas, permitindo que o usuário extraia o melhor de cada um.**

A popularização de um padrão de conexão e comunicação entre componentes tornará os pacotes de *software*, baseados neste tipo de componentes, fontes ricas de recursos. Alguns de seus componentes podem ser reestruturados conforme uma necessidade específica, ou podem ser reaproveitados em outras aplicações. Outra abordagem, proposta pelo ESCOT, é que pacotes de *software* e ferramentas de produção sejam capazes de produzir seus resultados na forma de componentes ou coleções de componentes, de modo que o produto final possa ser integrado com outras produções.

Isto irá propiciar uma oportunidade única de combinar diferentes gêneros de ferramentas de aprendizagem que são mais valiosas combinadas que individualmente. Este tipo de combinação entre ferramentas irá implementar mais efetivamente abordagens de aprendizagens inter-disciplinares e holísticas [ROS99].

Do ponto de vista do aluno:

- **Utilizar componentes de *software* dispostos em um ambiente visual como blocos de construção promove o desenvolvimento de atividades, onde o aluno atua como agente ativo na construção do conhecimento.**

Ao conceber o Boxer, Andrea diSessa antecipou diversos benefícios de um ambiente visual baseado em componentes de *software*. De fato, atualmente ele está envolvido no projeto Web/comp [DIS01], que se propõe a desenvolver um estudo prático do uso de componentes de *software* em atividades de ensino-aprendizagem.

Existem diversas estratégias para explorar componentes de *software* em atividades de ensino-aprendizagem. Entre elas estão os ambientes visuais que disponibilizam para o aprendiz coleções de componentes, usados como blocos de construção no desenvolvimento de projetos com propósito educacional, como simulações, animações, apresentações, etc. Neste tipo de atividade o aluno estrutura o projeto ao seu modo, configurando e combinando componentes livremente. Cada solução pode ser interativamente testada e sofrer refinamentos sucessivos até que se alcance o objetivo desejado.

Ao mesmo tempo em que os componentes são capazes de oferecer aos alunos um sistema mais flexível que a maioria dos pacotes de *software*, não limitando sua intervenção à definição ou modificação de parâmetros, eles podem se tornar quase tão versáteis quanto as linguagens de programação, proporcionando um ambiente mais propício à prática de projetos educacionais.

Coleções de componentes previamente confeccionados ou selecionados para um projeto podem oferecer versatilidade, sem a complexidade exigida por uma linguagem de programação, ainda que componentes deste tipo não sejam tão versáteis quanto o uso da linguagem. Isto pode ser compensado se a estrutura do componente puder ser modificada pelo aluno. Diversos ambientes já apresentam mecanismos ideais para a definição do comportamento de componentes, entre eles está o Boxer [DIS97] e a linguagem *Visual Agent Talk* [REP96] que será apresentada mais adiante.

- **O intercâmbio de componentes facilita a colaboração entre alunos dentro de um mesmo grupo ou entre grupos. Aqui a *Web* também cumpre um papel importante como mediadora, para uma comunicação que pode estender-se a um âmbito global. Além disto, ela cumpre o papel de repositório de componentes.**

Diversas características de ambientes baseados em componentes estimulam o intercâmbio entre autores: os componentes são fáceis de transportar, composições compartilhadas, baseadas em componentes, são fáceis de analisar e adaptar (além de permitir a extração e reaproveitamento de seus componentes), os componentes podem ser utilizados e configurados sem conhecimentos específicos de sua implementação e finalmente, os componentes podem ser projetados como elementos autônomos, que se integram com facilidade em novos ambientes.

Além da facilidade de troca dos componentes de *software* individualmente, produções inteiras podem ser compartilhadas e aproveitadas, integral ou parcialmente, por outros grupos. Isto se deve à simplicidade de compreender a estrutura de uma composição feita com componentes e a facilidade em extrair fragmentos da mesma.

Como já foi abordado anteriormente, a estrutura de objetos e componentes de *software* é ideal para intercâmbio através da *Web*. Um exemplo bastante interessante deste tipo de abordagem é o *Visual AgenTalk Behavior Exchange* [REP96], que consiste em um ambiente que utiliza a *Web* como meio para que os alunos possam trocar desde produções completas, até componentes individuais ou apenas um comportamento associado a um deles.

2.2.3. Utilizando objetos e componentes em projetos educacionais

Os benefícios apresentados têm motivado o surgimento por todo o mundo de vários projetos, que adotam objetos e componentes de *software* em atividades de ensino-aprendizagem.

No Apêndice E.3 – **Projetos educacionais baseados em objetos e componentes** – apresentamos alguns desses projetos, suas diferentes abordagens e contribuições. Cada um lança uma proposta singular no que diz respeito ao papel dos objetos e componentes no desenvolvimento da atividade. Sua análise nos propicia vislumbrar um amplo panorama de possibilidades e reconhecer algumas demandas de alta prioridade, tal como a configuração de um padrão que permita a integração dos produtos.

A tarefa de estabelecer um padrão de integração nos remete novamente ao paradigma de orientação a objetos. Este paradigma assumiu múltiplas formas nas diversas áreas da computação, encontrando, para cada uma, novas abordagens e benefícios. Eles têm sido especialmente interessantes em contextos onde se apresentam diferentes configurações de equipamentos ou *software*, que necessitam comunicar-se e trocar dados. Isto se deve, principalmente, à capacidade dos objetos de esconder as particularidades de sistemas diversos em cápsulas de software, que deixam transparecer apenas os canais necessários para a sua comunicação (interface), que é feita através de mensagens. As mensagens independem de equipamentos ou sistemas específicos e podem, deste modo, circular livremente entre ambientes heterogêneos. Como destaca Frank Manola: "Há uma crescente concordância que modelar um sistema distribuído como uma coleção distribuída de objetos interativos provê uma *framework* apropriada para ser usada na integração de recursos computacionais heterogêneos, autônomos e distribuídos (HAD)". [MAN98]

No Apêndice E.4 – **Integrando e Colaborando** – são apresentadas iniciativas que, partindo da singular capacidade de integração dos objetos, propõem a elaboração de um padrão no contexto educacional, de modo que objetos e componentes de *software* de origens diferentes possam se comunicar e possam ser livremente intercambiados, principalmente através da *Web*.

2.3. Objetos de conhecimento e metadados

Com a expectativa de um grande crescimento de objetos e recursos a ser disponibilizados através da *Web*, surge um outro desafio: criar um sistema eficiente e padronizado de classificação para esta enorme quantidade de objetos distribuídos por todo o mundo.

Para atender a esta demanda emergente, o IEEE *Learning Technology Standardization Committee* (LTSC)[3] vem liderando um projeto para a elaboração de um padrão de metadados para "objetos de conhecimento", o LOM - *Learning Object Metadata* [LTS00].

Estes metadados descrevem os objetos de conhecimento, ou melhor, são dados que descrevem outros dados (metadados) sob a forma de objetos de conhecimento.

Diversos esforços têm sido realizados para se estabelecer vocabulários comuns de metadados, através da padronização de conjuntos de descritores, interpretados de forma única

[3] A página do LTSC pode ser encontrada no endereço – http://ltsc.ieee.org/.

entre as diversas comunidades. Estes padrões promovem a interoperabilidade entre sistemas e comunidades diversos. Em particular, pode ser destacada a iniciativa Dublin Core [WEI95].

Dublin Core constitui um conjunto de metadados com o objetivo original de permitir que autores agreguem uma descrição a seus recursos produzidos para *Web*. Esta descrição facilita a exploração de recursos eletrônicos. Tal iniciativa atraiu a atenção de comunidades que fazem uso da descrição formal de recursos, como museus, bibliotecas, agências governamentais e organizações comerciais.

Na área educacional, as iniciativas na elaboração de conjuntos de metadados para uma estrutura descritiva de recursos educacionais são principalmente abordadas no Projeto ARIADNE[4], no *Learning Object Metadata* (LOM), e no IMS *Learning Resource Meta-data* [AND00].

Os três projetos estão bastante inter-relacionados e apóiam-se à definição de Objeto de Conhecimento (*Learning Object*) definido pelo IEEE:

> Objeto de conhecimento é aqui definido como qualquer entidade, digital ou não digital, que pode ser utilizada, reutilizada ou referenciada durante o aprendizado apoiado sobre a tecnologia. Exemplos de aprendizado apoiado em tecnologia incluem sistemas de treinamento baseados no computador, ambientes de aprendizado interativo, sistemas inteligentes de instrução auxiliada por computador, sistemas de aprendizado à distância, sistemas de aprendizado baseados na *Web* e ambientes de aprendizado colaborativo [LTS00].

O termo ‘objeto’ é utilizado de forma bem mais ampla que a estabelecida pelo paradigma da orientação a objetos, referindo-se a entidades digitais e não digitais. De qualquer modo, por ser mais abrangente, aplica-se igualmente ao modelo de objetos apresentado.

Os metadados que descrevem esses objetos são definidos em termos de propriedades e valores. Cada objeto possui um conjunto de propriedades a ele relacionadas como: assunto, data de criação, etc. Uma instância específica desses objetos possui valores para cada propriedade, por exemplo:

- Assunto = “Usando componentes”
- Data de criação = “15/07/2000”

Para padronizar as propriedades que descrevem cada objeto de conhecimento, no sentido de viabilizar o intercâmbio dos mesmos, a especificação adota o conceito de esquemas (*schemes*). Os esquemas são estruturas hierárquicas que determinam quais são as propriedades

[4] A página do projeto ARIADNE pode ser encontrada no endereço – http://ariadne.unil.ch/.

associadas a um objeto de conhecimento, definindo para cada um o seu tipo, domínio e obrigatoriedade.

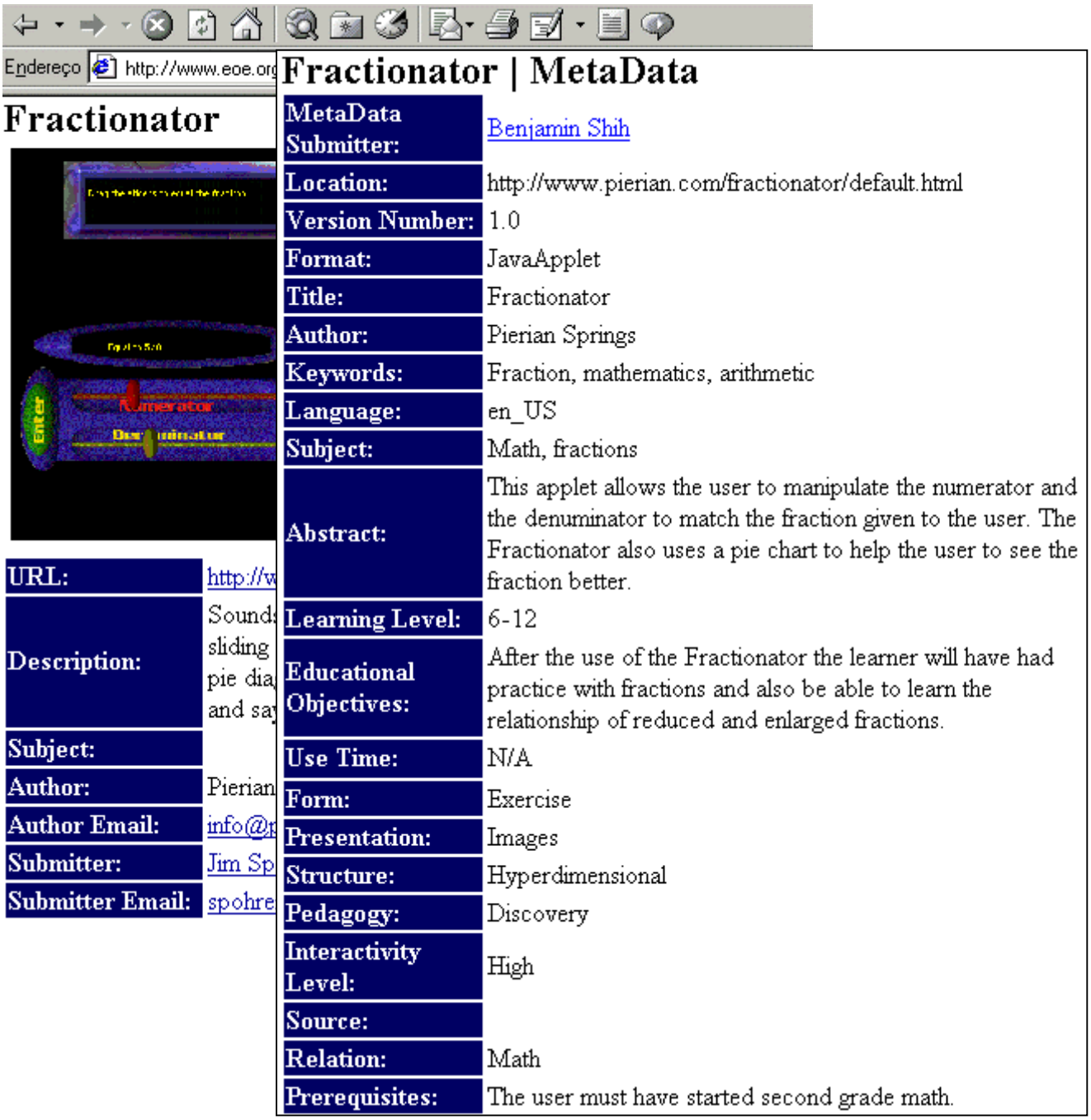
O padrão do IEEE é montado sobre um esquema bastante genérico, denominado BaseScheme (esquema básico), cujo princípio é reunir os principais elementos comuns entre os objetos educacionais. Este esquema pode ser estendido para outros esquemas derivados, de acordo com necessidades específicas.

A estrutura geral do BaseScheme é dividida segundo grupos de propriedades:

General	Dados de identificação, chaves de acesso, delimitação do domínio, descrição do conteúdo e da estrutura.
LifeCycle	Dados para controle e documentação do ciclo de vida do documento.
MetaMetaData	Referências à origem e estrutura dos metadados.
Technical	Dados técnicos, tais como: formato, tamanho, requisitos de sistema operacional, duração, etc.
Educational	Elementos de descrição pedagógica do recurso, tais como: abordagem, nível de interatividade, pré-requisitos, objetivo educacional, grau de dificuldade, tempo esperado de trabalho, etc.
Rights	Dados referentes às condições de uso do produto e, eventualmente, valores a serem pagos pelo uso do recurso.
Relation	Características do recurso em relação a outros recursos.
Annotation	Comentários referentes ao uso educacional do produto.
Classification	Referência que determina onde o recurso será colocado dentro de um sistema de classificação específico.

A estrutura da especificação de metadados do IMS é baseada no LOM, com algumas modificações. Adicionalmente, o IMS define uma representação em linguagem XML para os metadados.

Na Figura 2-2, observamos uma aplicação prática do padrão IMS realizada pelo EOE, que permite a associação de metadados, conforme este padrão, aos objetos de conhecimento de seu repositório.



Endereço http://www.eoe.org

Fractionator

URL:	http://w
Description:	Sound sliding pie dia and sa
Subject:	
Author:	Pierian
Author Email:	info@p
Submitter:	Jim Sp
Submitter Email:	spohre

Fractionator | MetaData

MetaData Submitter:	Benjamin Shih
Location:	http://www.pierian.com/fractionator/default.html
Version Number:	1.0
Format:	JavaApplet
Title:	Fractionator
Author:	Pierian Springs
Keywords:	Fraction, mathematics, arithmetic
Language:	en_US
Subject:	Math, fractions
Abstract:	This applet allows the user to manipulate the numerator and the denuminator to match the fraction given to the user. The Fractionator also uses a pie chart to help the user to see the fraction better.
Learning Level:	6-12
Educational Objectives:	After the use of the Fractionator the learner will have had practice with fractions and also be able to learn the relationship of reduced and enlarged fractions.
Use Time:	N/A
Form:	Exercise
Presentation:	Images
Structure:	Hyperdimensional
Pedagogy:	Discovery
Interactivity Level:	High
Source:	
Relation:	Math
Prerequisites:	The user must have started second grade math.

Figura 2-2: Ficha com metadados padrão IMS, associada a um objeto no repositório EOE.

Autores de *software* e usuários podem contribuir com estes metadados, o que torna a tarefa de documentação do *software* disponível uma atividade pública e participativa. A base de indexação resultante deste processo permitirá, cada vez de forma mais completa, encontrar produtos de *software* de acordo com demandas bastante específicas, tais como: sistemas de simulação de um tema de física para uma série escolar determinada, uma animação que ilustre de forma tridimensional a duplicação do DNA, e assim por diante.

2.4. Objetos e componentes no sistema Casa Mágica

O sistema Casa Mágica[5] [SAN98] constitui-se em um ambiente para construção de aplicações educacionais. Ele combina os recursos de uma ferramenta de autoria com recursos que permitem a construção e exploração de modelos de estudo. Sua estrutura para a construção de aplicações educacionais é bastante flexível, tornando o sistema Casa Mágica apropriado para qualquer nível escolar ou domínio de conhecimento. Uma das características do sistema que contribui para esta flexibilidade é a possibilidade de expandir suas bibliotecas de recursos e configurar suas ferramentas para atuarem em domínios específicos.

O sistema inteiro está fundamentado sobre uma estrutura de objetos e componentes de *software*; isto permitiu que este ambiente fosse utilizado como base para o desenvolvimento do projeto *Anima*.

A seguir, introduziremos os conceitos fundamentais do sistema Casa Mágica, a fim de estabelecer claramente o contexto dentro do qual *Anima* foi concebido e implementado.

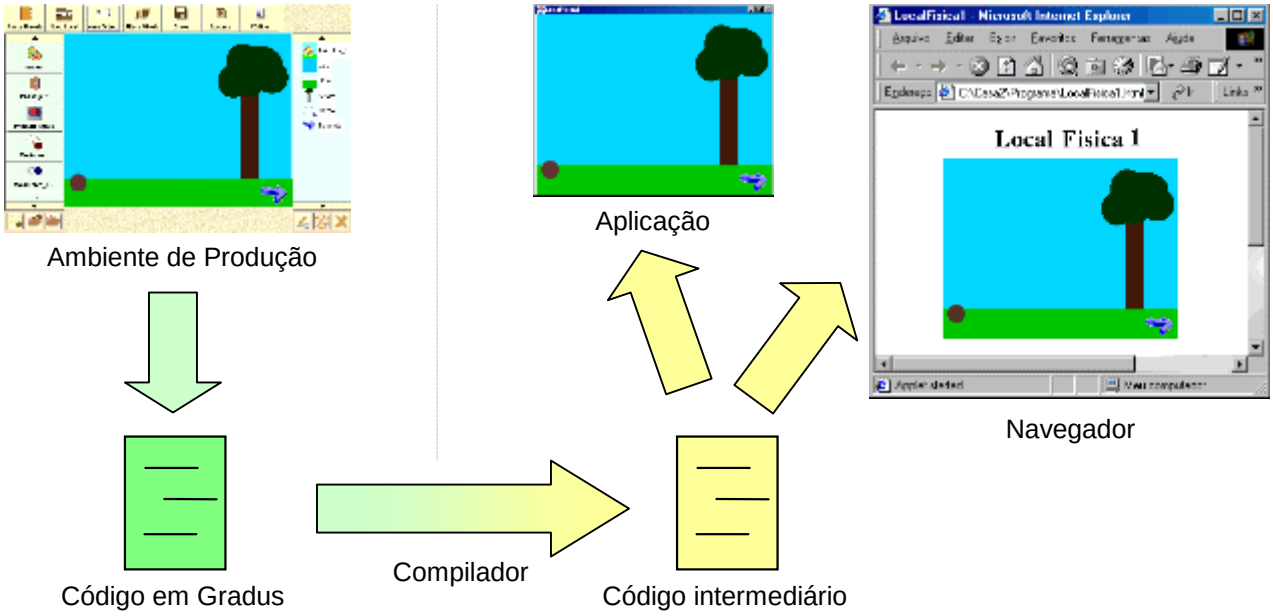


Figura 2-3: Diagrama de funcionamento de uma aplicação produzida em Casa Mágica.

Como pode ser observado no diagrama da Figura 2-3, o sistema é constituído de um conjunto de módulos interligados em um ambiente integrado de produção e execução. Todo o material elaborado no Ambiente de Produção é codificado em uma linguagem própria (Gradus) e compilado para um código intermediário. Uma parte deste código é gerada

[5] O Sistema Casa Mágica, apresentado neste trabalho, consiste na versão 2.0 do programa. A versão 1.0 foi vencedora do I Concurso Nacional de Software Educacional e Tecnológico promovido pelo MEC em 1995. Ambas as versões são resultados de trabalho desenvolvido pelo autor desta dissertação.

automaticamente por sistemas de produção e exploração visuais e outra parte é diretamente codificada pelo autor.

Na versão original do sistema Casa Mágica, uma aplicação era constituída basicamente pelo código intermediário e pelos dados de mídia por ele usados. Este código era tratado por um interpretador, executado diretamente a partir do sistema operacional ou dentro de um navegador *Web*. Posteriormente, com a introdução de *Anima* dentro do sistema, uma grande parte da aplicação passou a ser representada utilizando XML. Isto será aprofundado posteriormente, no capítulo 4.

Uma aplicação completa em Casa Mágica consiste em um conjunto de Locais interligados entre si. Um Local representa uma região, concreta ou abstrata, onde se desenvolve a aplicação propriamente dita. Em cada Local é montado um cenário. Os Locais são interligados através de portas (que se comportam como hiperligações e podem ter qualquer aparência), como pode ser observado na Figura 2-4.

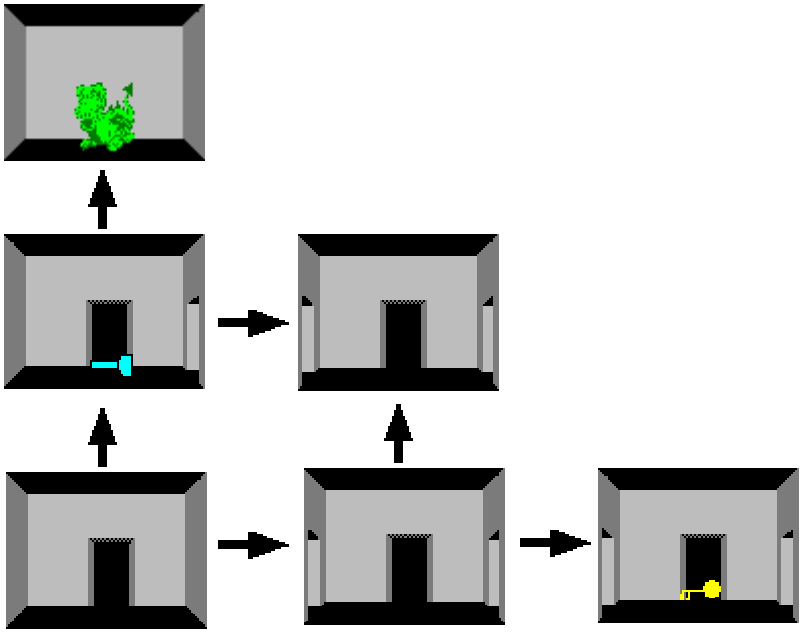


Figura 2-4: Conjunto de Locais de uma aplicação interligados por portas.

Cada um dos Locais é formado por objetos interligados entre si (Figura 2-5). Os objetos correspondem a modelos de qualquer elemento concreto ou abstrato.

Dentre os objetos disponíveis, estão aqueles que podem ser configurados e interligados visualmente, através de um conjunto de ferramentas do sistema. Estes objetos constituem uma categoria especial de componentes.

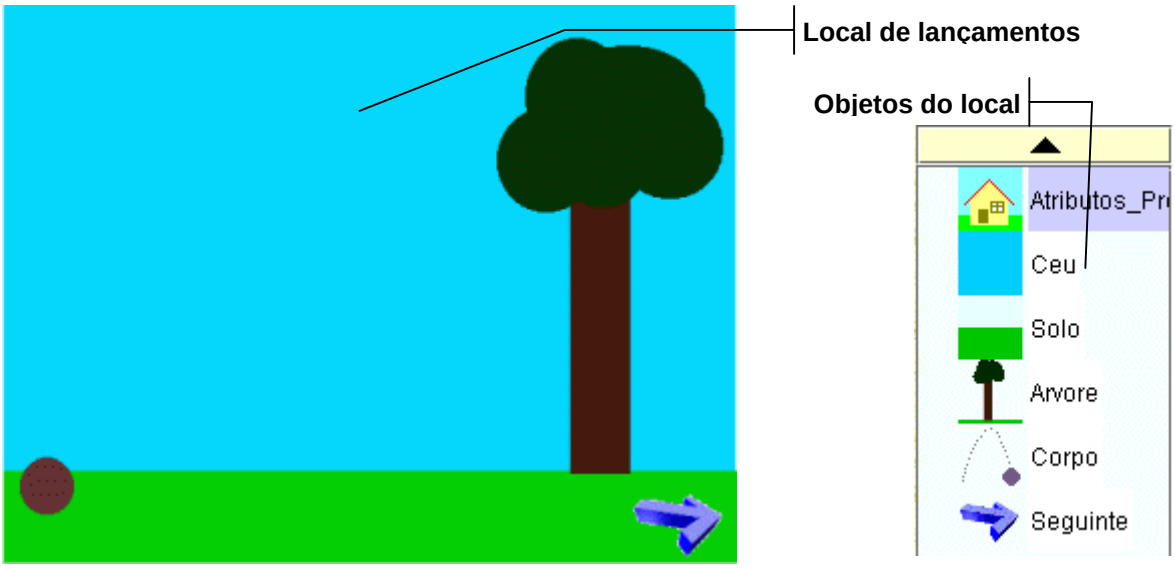


Figura 2-5: Local de lançamento de um corpo (projeto de física) e barra mostrando seus objetos.

Durante a construção ou modificação de locais, o sistema abre uma barra onde estão dispostos os componentes disponíveis no sistema (Figura 2-6). Os componentes podem ser agrupados por tema, de modo que sejam apresentados aos usuários apenas aqueles pertinentes à atividade que irá desenvolver.

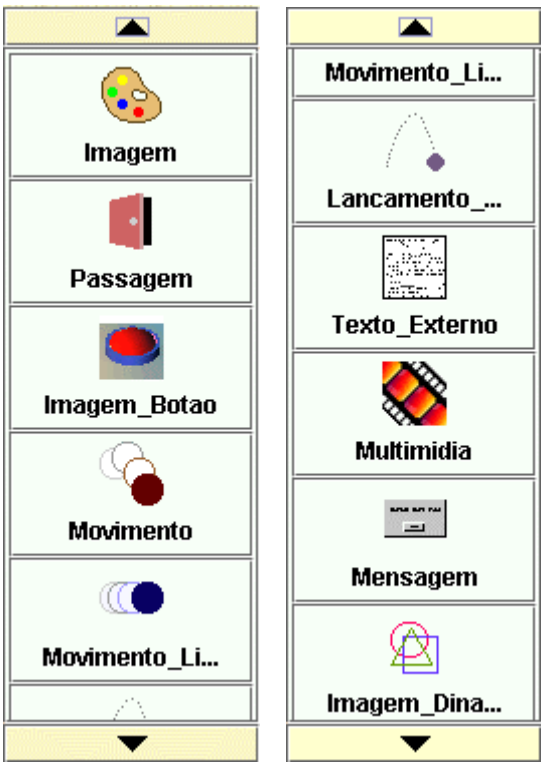


Figura 2-6: Barra de componentes do sistema Casa Mágica.

A configuração dos componentes é feita visualmente, através de painéis denominados Visores. Através deles é possível modificar propriedades destes componentes, tais como aparência, parâmetros de comportamento, etc.

Os objetos comunicam-se entre si através de mensagens. Através de um sistema de conexão visual, o ambiente de produção permite que um componente seja configurado para enviar mensagens para outro.

Estas mensagens são enviadas na medida em que ocorrem determinados “eventos”, tais como: uma ação do usuário (ex: mouse clicado, tecla pressionada, etc.), uma operação que se completa (ex: corpo alcança uma meta, calculadora termina um cálculo, etc.), um evento temporal (ex: tempo transcorrido em um relógio), etc..

Na Figura 2-7 observamos o processo de associação de um componente botão a um componente relógio. Durante a associação, o sistema solicita o evento que dispara a mensagem (neste caso, será o botão clicado pelo mouse) e a mensagem a ser enviada para o relógio (neste caso, será uma mensagem para ativação do mesmo).

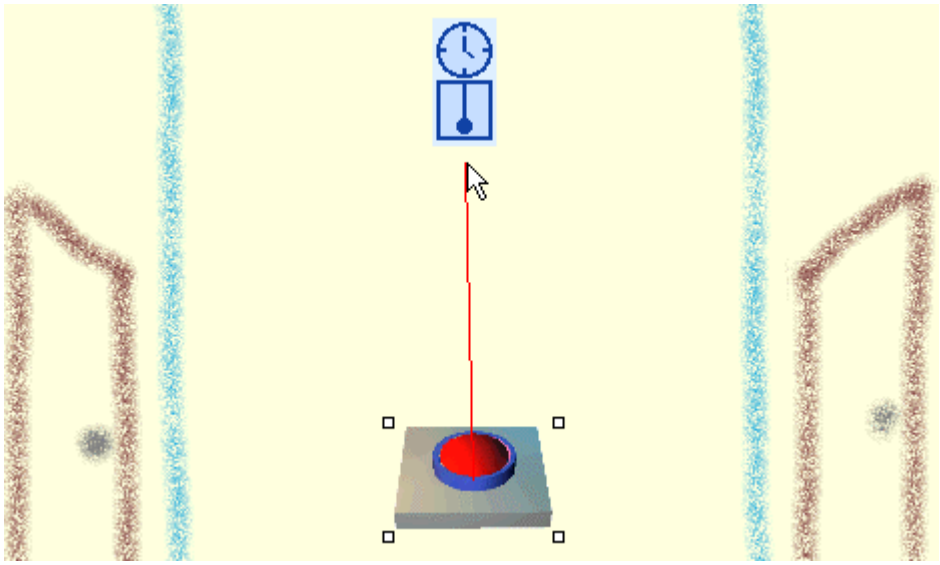


Figura 2-7: Componente botão é configurado para enviar mensagem para componente relógio.

A produção resultante, formada de objetos interligados por mensagens, pode ser concebida como uma rede, a qual usualmente chamamos de composição.

2.4.1. A produção como objeto de estudo

Além de serem os tijolos básicos na montagem dos projetos, os componentes podem ser tomados como objetos de estudo. Para tanto, o sistema Casa Mágica dispõe de ferramentas que permitem a exploração da sua concepção, através da análise e modificação de suas propriedades e seu comportamento. Isto é feito através de duas ferramentas: os visores e o laboratório.

Os visores são projetados pelo professor de acordo com a perspectiva na qual ele espera que o aluno veja seu objeto de estudo. Para uma mesma classe de objetos, pode haver múltiplos visores, que apresentam as propriedades mais relevantes para o tipo de análise pretendida. Estes visores podem ser ajustados para diferentes níveis de detalhamento de um modelo (para diferentes séries, por exemplo). Na Figura 2-8 são apresentados dois visores, com níveis de detalhe diferentes, para um objeto de física que simula um lançamento oblíquo (uma imagem do local é apresentada na Figura 2-5).

No laboratório, tanto o professor quanto o aluno, durante o processo de confecção dos modelos, podem testar os objetos produzidos, sem que haja qualquer interferência no restante da aplicação. Todos os comandos e parâmetros utilizados são registrados, produzindo um histórico de todas as atividades. Esta prática é bastante útil para que o professor possa acompanhar a atividade de seus alunos, e para que estes possam rever qualquer um de seus procedimentos.

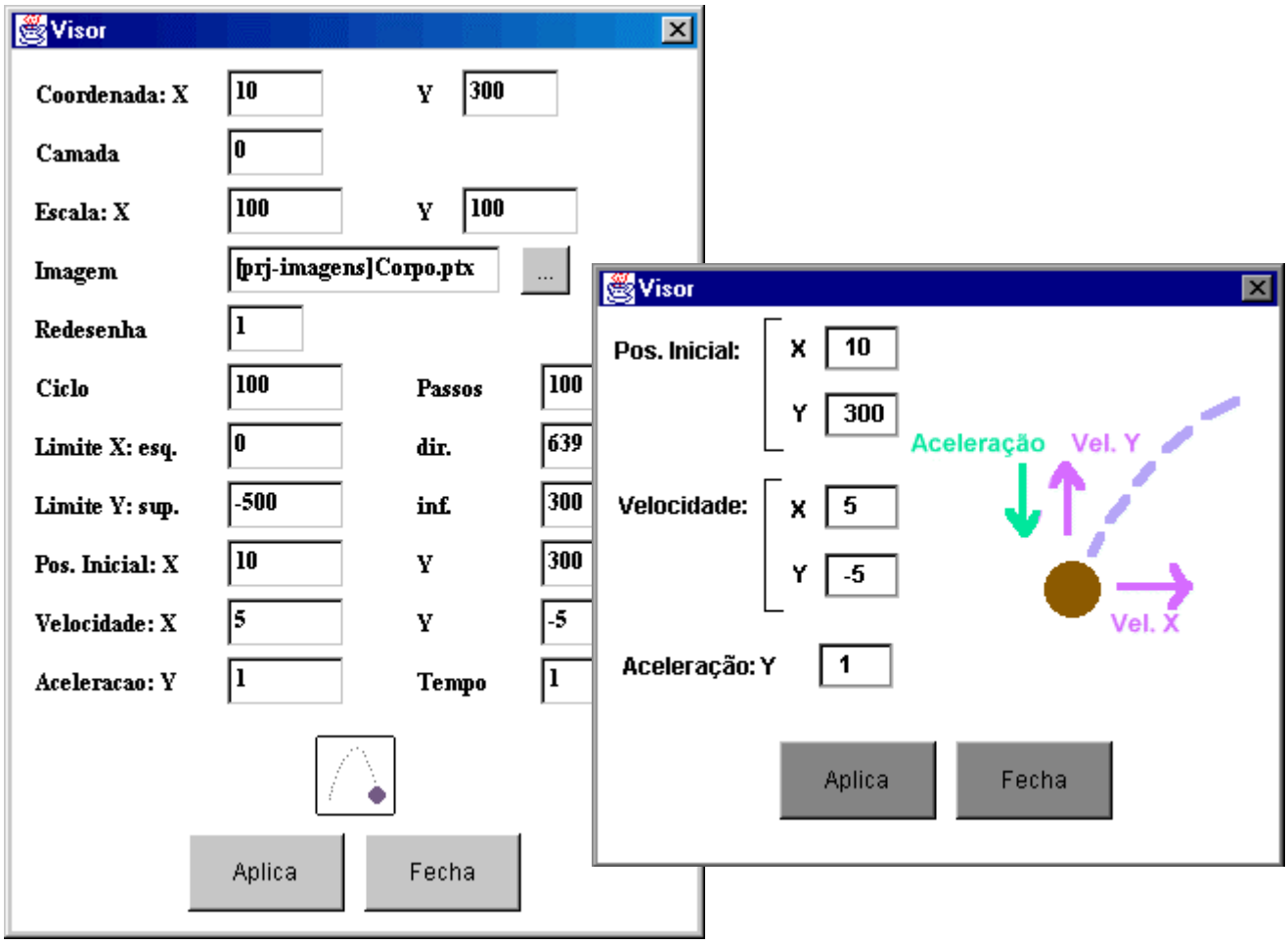


Figura 2-8: Visores alternativos para o mesmo componente – corpo que descreve trajetória de lançamento oblíquo.

No exemplo apresentado na Figura 2-9, um componente denominado ‘Forma’, capaz de traçar formas geométricas, foi colocado sobre a imagem de um quadro. O módulo de laboratório foi ativado e, a partir de instruções enviadas para o componente, foi desenhado o gráfico que está sobre o quadro. Observe-se que no painel do laboratório foi inserido o histórico da operação, indicando inclusive se o aluno comete algum engano (veja mensagem de advertência em destaque “Ação flutua não existe para o objeto Forma”).

Criar meios para que os componentes se transformem em objeto de estudo, estabelece uma outra perspectiva no desenvolvimento de projetos educacionais, na qual não apenas o produto final é importante, como também sua estrutura de construção.

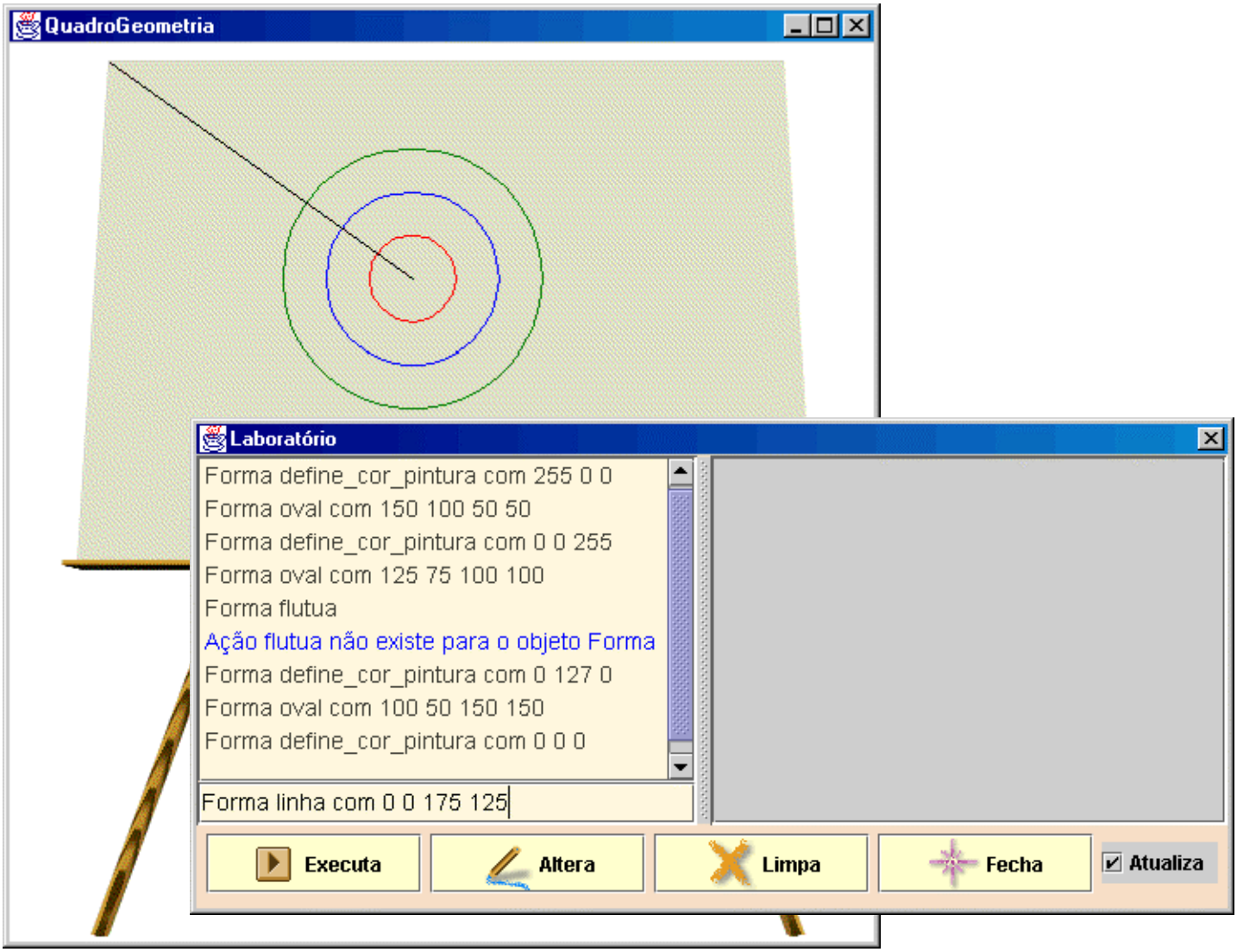


Figura 2-9: Interação com o componente “Forma” através do painel de laboratório.

Conclusão

Neste capítulo obtivemos uma visão abrangente referente ao uso da tecnologia de objetos e componentes de *software* na educação. Partimos de uma análise da problemática, passando por uma introdução aos fundamentos de objetos e componentes de *software*, cuja abordagem foi confrontada com as tecnologias tradicionais. Finalizamos apresentando projetos desenvolvidos, onde os objetos e componentes assumem diferentes papéis e, em especial, um projeto desenvolvido localmente pelo autor deste projeto, o sistema Casa Mágica.

Deste modo, este capítulo cumpre seu objetivo de contextualizar o projeto *Anima* dentro do cenário de aplicação dos objetos e componentes de *software* e atividades de ensino-aprendizagem. O próximo capítulo é dedicado aos aspectos tecnológicos que envolvem a aplicação do modelo de objetos ao ambiente *Web*.

3. MODELOS DE OBJETOS PARA *WEB*

Este capítulo aborda o uso da tecnologia de objetos no desenvolvimento de *software* para *Web*. Além disso, analisa técnicas associadas, como as Linguagens de Código Móvel (*Mobile Code Languages* – MCL) e um conjunto de linguagens de marcação estruturada baseadas em XML.

A partir deste estudo, obtemos um quadro referencial do conjunto de tecnologias sobre as quais está estruturado o projeto *Anima*. O capítulo está dividido em dois tópicos:

O tópico **Fundamentos de um Modelo de Objetos para *Web*** aborda o "Modelo de Objetos", utilizado por Frank Manola, visando caracterizar modelos utilizados por sistemas baseados em objetos e definir um modelo de objetos adequado para *Web* (*Web Object Model*) [MAN99], cujos princípios são utilizados no modelo de *Anima*.

O tópico seguinte, **Representação de Objetos através de Marcação Estruturada**, aborda a representação de objetos de *software* utilizando linguagens de marcação estruturada.

3.1. Fundamentos de um Modelo de Objetos para *Web*

A partir de princípios estruturados inicialmente pela Programação Orientada a Objetos, no final da década de 60, muitas outras áreas desenvolveram, com maior ou menor grau de compatibilidade, mapeamentos de tais princípios, encontrando em seus domínios peculiaridades para as quais os objetos apresentavam contribuições significativas.

Este foi o caso das redes de computadores, onde o intercâmbio e integração dinâmica dos dados de máquinas distintas incentivaram a construção de objetos que se intercomunicam e trabalham cooperativamente através da rede. Simultaneamente, foram criados mecanismos para que os objetos pudessem trafegar pela rede, promovendo sua transferência integral de um ponto a outro.

Isso se deve, principalmente, à capacidade dos objetos de encapsular as particularidades de sistemas diversos e de comunicar-se através de mensagens que dependem exclusivamente de suas interfaces.

Se isto foi especialmente interessante para interligar sistemas de empresas, seja dentro de uma rede local, seja entre edifícios fisicamente distantes, com a popularização da Internet,

dotada de uma grande heterogeneidade de plataformas de hardware e sistemas, os objetos se mostraram especialmente promissores.

O encontro entre a Internet e os objetos processa-se gradativamente. Diversas linguagens e padrões para a integração de objetos, tal como CORBA, têm buscado caminhos que os tornam mais compatíveis e, conseqüentemente, mais integrados com a Internet, particularmente com a *Web*, que é sua principal aplicação.

Dentre as principais mudanças está a adaptação do formato utilizado para representação e transmissão dos dados, de modo que se utilizem aqueles padrões que se têm estabelecido na *Web*, tais como XML [BRA00] e suas derivações (aplicações XML).

Além disto, a difusão de Linguagens de Código Móvel (*Mobile Code Languages* – MCLs) – cujos programas podem ser transmitidos através da rede e executados em máquinas de diferentes arquiteturas e sistemas operacionais, devido à sua independência – tem conduzido a uma concepção de aplicações nas quais os objetos são transmitidos pela rede e executados diretamente na máquina do cliente, seja ele quem for.

Segundo Alfonso Fuggeta, "Mobilidade de código pode ser definida como a capacidade de modificar dinamicamente as ligações entre fragmentos de código e o local onde eles são executados" [FUG98]. Esta capacidade permite que programas possam trafegar livremente pela Internet, apesar de sua grande diversidade de plataformas e sistemas.

Por exemplo, Java é uma linguagem que faz uso de código móvel. Seus programas são compilados para um formato intermediário denominado *bytecode*. Este formato não está associado a qualquer arquitetura de computador ou sistema operacional específico. Tendo uma concepção bastante genérica, os *bytecodes* Java são executados através de módulos *runtime*, responsáveis por realizar esta ligação dinâmica entre os fragmentos de código, e a plataforma e o sistema operacional onde ele está sendo executado. Deste modo o código se torna 'móvel' entre plataformas e sistemas, pois será sempre o mesmo código a ser executado em todas elas, o que muda é a parte responsável por ligá-lo ao local de execução (*runtime*).

A crescente possibilidade de distribuir programas além das mídias habituais da *Web* – tal como texto, imagem, vídeo e animação – conduz a uma nova ótica no desenvolvimento de aplicações para a rede. Uma significativa parcela destas MCLs – tal como Java – adota o modelo de objetos e distribui seu código sob a forma de "objetos móveis".

Em paralelo com a difusão das MCLs, os atuais padrões de representação e transmissão de dados da Internet estão sendo revistos. A linguagem HTML, uma das

principais responsáveis pela difusão da *Web*, tem se apresentado limitada para novas exigências, impostas em situações como:

- Diversas organizações têm adotado a Internet como meio de integração entre estabelecimentos fisicamente distantes, e a *Web* como veículo de comunicação com a comunidade (clientes, fornecedores, etc.). A linguagem HTML cumpre bem o papel de apresentação e estabelece um nível básico de interação com o usuário, mas o restante do processo acaba ficando por conta de linguagens e mecanismos que giram em torno dela.
- A *Web* tem demandado linguagens especializadas para formas particulares de expressão, como apresentações multimídia, expressões matemáticas ou estruturas químicas.

Isto vale também para aplicações baseadas em objetos, que têm demandado uma revisão nos modelos de representação de dados, de modo a se tornarem mais integrados com aqueles utilizados por sistemas orientados a objetos. Isto significa a constituição de um modelo de objetos para *Web*.

Em abril de 1992 formou-se o Comitê Técnico H7[1] (*Object Information Management*) pertencente ao *National Committee for Information Technology Standards* (NCITS)[2] [MAN97] com os seguintes propósitos:

- "Baseado nas recomendações do *Object-Oriented Database Task Group* (OODBTG), desenvolve um relatório técnico para refinar mais o modelo referencial de gerenciamento de informações baseado em objetos".
- "Trabalha ativamente com outras organizações para adquirir consenso e harmonia em áreas potenciais para padronização de objetos" [MAN97].

O Comitê adotou o termo *Object Model* (Modelo de Objetos) para caracterizar os modelos subjacentes associados a linguagens, metodologias de especificação, análise ou projeto orientado a objetos [MAN97]. Ele destaca, em seu relatório, que este termo também é utilizado para descrever coleções de objetos criados para modelar um sistema particular ou aplicação [MAN97] como, por exemplo, o *Document Object Model* (DOM) [WOO98], que

---

[1] A página do Comitê Técnico H7 pode ser encontrada no endereço http://www.objs.com/x3h7/h7home.htm.

[2] NCITS – *National Committee for Information Technology Standards* – era a denominação em 1992 do atual INCITS – *InterNational Committee for Information Technology Standards* – cuja missão é produzir padrões de mercado na área de Tecnologia da Informação, a partir de livre consenso. A página do INCITS pode ser encontrada no endereço http://www.x3.org/incits/.

estabelece um modelo associado a documentos XML e HTML, na forma de uma coleção de objetos [MAN99]. Como se pode observar, esta segunda interpretação é bem diferente da primeira e não caracteriza o conceito que é adotado pelo Comitê e que utilizaremos no restante deste trabalho.

Como resultado da análise de diversos modelos de objetos, o Comitê Técnico H7 produziu uma matriz (*Object Model Features Matrix*) que apresenta diversas linguagens e especificações, confrontando seus modelos.

Frank Manola, editor do Comitê H7, utiliza o conceito de *Object Model* em um estudo referente a modelos de objetos adequados para *Web* (*Web Object Model*) [MAN99]. Ele destaca três aspectos fundamentais para qualquer modelo de objetos:

- Estruturas de dados que representam o estado do objeto.
- Meios para associar comportamento (métodos do objeto) ao estado do objeto.
- Meios para os métodos do objeto acessarem e operarem as variáveis que representam o seu estado.

Como será apresentado adiante, o projeto *Anima* implementa um modelo de objetos móveis, adequado para *Web* e independente de plataforma. Para alcançar estes objetivos, o modelo adota as linguagens padronizadas XML e RDF, e contempla os três aspectos mencionados por Manola [MAN99]. Sua estrutura inclui o uso de MCL na constituição de objetos móveis.

3.1.1. Representação de um objeto

A fim de estruturar a análise feita adiante, referente a modelos de objetos, criamos a seguinte classificação para os elementos que compõem um objeto:

- **Elementos Dinâmicos** – estão associados a cada objeto individualmente. Podemos destacar dois tipos de elementos dinâmicos: a identidade única que individualiza o objeto dentro de seu universo, e seu estado interno, definido por atributos permanentes ou transitórios, tais como: cor, tamanho, etc.
- **Elementos Estáticos** – são compartilhados por todos os objetos que pertencem a uma mesma classe, tais como métodos, herança, etc. Os elementos estáticos, através dos métodos e características da classe, definem o comportamento dos objetos pertencentes à classe.

Como é evidenciado no relatório *Object Model Features Matrix*, existe uma ampla variedade de modelos de objetos utilizados pelas diferentes linguagens e especificações. Em especial, é importante se observar a diferença entre modelos baseados em classes e modelos baseados em protótipos.

Nos modelos baseados em classes, tais como Smalltalk e Java, os objetos são instâncias de classes definidas separadamente, deste modo, os elementos estáticos são definidos dentro da classe, sendo compartilhados por todos os objetos da mesma classe, e os elementos dinâmicos são definidos em cada objeto. Por outro lado, nos modelos baseados em protótipo, tal como o WCML [GEL99], não existe a distinção entre classe e objeto. Os objetos reúnem os elementos estáticos e dinâmicos e a herança é definida diretamente entre objetos, ao invés de classes.

Por realizarem a separação entre classe e objeto, os modelos baseados em classes permitem estabelecer uma divisão entre elementos estáticos e elementos dinâmicos.

O Smalltalk, e muitas outras linguagens de programação orientadas a objetos adotam a seguinte representação para os elementos estáticos e dinâmicos dos objetos:

- **Elementos Dinâmicos** – o estado interno é definido a partir de valores de atributos associados ao objeto. Quando o objeto se encontra na memória, os atributos podem ser representados sob a forma de variáveis.

  A classe estabelece quais os atributos que descrevem um objeto; cada objeto possui valores de estado específicos para os mesmos. Deste modo, é o objeto que determina os valores de estado propriamente ditos e é, a partir deles, além da identidade, que objetos da mesma classe se diferenciam.

  A identidade é definida através de um valor identificador exclusivo do objeto, geralmente um número.

- **Elementos Estáticos** – estão reunidos em um recurso independente do objeto, associado à classe.

  Os objetos fazem referência aos elementos estáticos da classe a que pertencem através de um ponteiro objeto→classe e, deste modo, são associados aos métodos e a outras características da classe, que afetam diretamente seu comportamento.

Enquanto as implementações das classes devem ser escritas em linguagens específicas de programação, a representação das suas instâncias não necessita do uso de instruções de

programação com sintaxe procedural. É suficiente apenas o registro dos valores das variáveis (propriedades) que representam o estado de cada instância. Esta separação é bastante propícia à serialização de objetos.

Jeff Johnson define serialização de um objeto como “colocar o objeto em uma forma que possa ser armazenado ou enviado como uma mensagem através de um cabo de rede” [JOH].

Mark Johnson define mais especificamente serialização como a conversão de um objeto sendo executado dentro de um programa, em uma seqüência de bytes que pode ser armazenada e/ou transmitida. Esta seqüência pode ser posteriormente utilizada na recomposição do objeto (o que significa recriar um objeto idêntico ao original) [JOH99.1].

Este não é um conceito novo e tem sido utilizado por diversos sistemas baseados em objetos, especialmente pelas linguagens de programação orientadas a objetos. Todavia, cada sistema tende a utilizar um formato particular de representação dos dados, geralmente incompatível com os demais.

A serialização pode registrar integralmente informações a respeito do objeto, ou seja, de seus elementos estáticos e dinâmicos, ou registrar apenas valores de estado do objeto e dados adicionais que permitam associá-lo a sua classe, como é comum nas linguagens orientadas a objetos. Isto é possível graças à separação mencionada anteriormente, que permite que a classe fique codificada em linguagem de programação em um recurso, e o objeto registre, em um recurso independente, apenas seus valores de estado e uma ligação para a classe. Este segundo recurso não precisa estar associado à linguagem de programação, por se tratar apenas de um registro de valores.

A necessidade de comunicação entre diversos sistemas baseados em objetos tem impulsionado a criação de padrões para serialização. Como será apresentado no próximo tópico, no domínio da Internet são especialmente relevantes alguns padrões que utilizam serializam os dados em XML.

3.2. Representação de Objetos através de Marcação Estruturada

Neste tópico analisamos propostas relevantes para a representação de objetos na *Web*, utilizando a linguagem XML. Cada uma delas se estrutura de maneira diversa, seja na forma como estabelece a marcação em XML, seja no modo como define e associa elementos estáticos e dinâmicos de um objeto.

Para comparar algumas características envolvidas com a representação de objetos, destacamos, na análise de cada padrão, três aspectos:

- **Elementos Dinâmicos:** formato utilizado para a representação dos elementos dinâmicos que caracterizam a instância de um objeto, e processo de mapeamento dos mesmos para um documento XML e vice-versa.
- **Elementos Estáticos:** estratégia para definição dos elementos estáticos que estabelecem o comportamento do objeto.
- **Implementação:** mecanismos utilizados para associar os elementos estáticos de um objeto à sua instância; executar seus métodos e permitir o acesso dos mesmos aos valores de estado do respectivo objeto.

A seguir apresentamos cada uma das propostas e ao final realizamos uma análise comparativa, enfatizando algumas características e limitações encontradas em seus modelos, que serviram como subsídio para o desenvolvimento de *Anima*.

OOXML

O OOXML [SIL98] define um padrão que integra a representação serializada do estado de um objeto e seu comportamento. Segundo Nicolas Silberzahn, OOXML é uma "perspectiva orientada a objetos da sintaxe *markup*" [SIL98] que se aplica a um recurso XML.

Elementos Dinâmicos

Cada elemento XML representa um objeto em OOXML. No exemplo a seguir temos quatro objetos: `Ficha`, `Nome`, `Idade` e `Sexo`.

```
<Ficha>
  <Nome>Mila</Nome>
  <Idade>21</Idade>
  <Sexo>Feminino</Sexo>
</Ficha>
```

Os elementos subordinados ao objeto `Ficha` (`Nome`, `Idade` e `Sexo`) representam Objetos Membro. Note que, neste caso, o elemento XML define a estrutura do objeto e os seus valores de estado, ou seja, ele determina quais são os objetos membro que descrevem o objeto `Ficha` e estabelece seus valores. Além do estado do objeto, os elementos também podem representar métodos.

Elementos Estáticos

Métodos são Objetos Membro, rotulados como `<Method>` que contém código Lisp descrevendo as ações do objeto. Os métodos são definidos dentro dos objetos, por este motivo cada objeto carrega seus próprios métodos.

```
<Ficha>
  <Nome>Mila</Nome>
  <Idade>21</Idade>
  <Sexo>Feminino</Sexo>

  <Method>
    <Name>View</Name>
    <Autor>Quincas</Autor>
    <Code>(lambda (this) (print "Informações do paciente"))</Code>
  </Method>
</Ficha>
```

No exemplo apresentado, o método **`View`** está associado ao objeto `Ficha` e contém instruções em Lisp para apresentar seu conteúdo quando for acionado. São aceitos métodos externos ao objeto, que são ligados ao mesmo através do elemento `<MemberOf>`.

Eventualmente, podem ser acrescentados ao método metadados, tal como o elemento `<Autor>`.

OOXML aceita a herança entre objetos, que é especialmente interessante quando vários objetos compartilham o mesmo comportamento. A herança é definida através do elemento `<isa>`. Objetos herdam características de outros objetos sem a necessidade de se criarem classes:

```
<Ficha>
  <isa>Registro</isa>
  <Nome>Mila</Nome>
  <Idade>21</Idade>
  <Sexo>Feminino</Sexo>
</Ficha>
```

No exemplo, o objeto `Ficha` é herdeiro do objeto `Registro`.

Além deste mecanismo de tornar um objeto específico herdeiro de outro objeto específico, OOXML define um mecanismo para se estabelecer a relação de herança externa a um objeto particular, que pode, por exemplo, tornar todos os objetos definidos como `Ficha` herdeiros do objeto `Registro`.

Implementação

Objetos em OOXML são apresentados a partir de um programa em código Lisp, capaz de realizar a leitura do arquivo XML que os contém e acionar qualquer dos métodos destes

objetos, uma vez que as rotinas Lisp embutidas nos mesmos podem ser lidas e interpretadas dinamicamente.

Uma das vantagens desta abordagem, conforme Nicolas Silberzahn, é o fato de se embutir nos dados seu próprio mecanismo de funcionamento e interpretação dos atributos [SIL98]. Vários objetos diferentes podem possuir o método `View` que apresenta seu conteúdo em uma saída padronizada. O programa acionará o método dos diversos objetos, sem se preocupar com o modo particular como cada um irá implementar a rotina. Desta maneira, ele poderá operar com novos objetos, que ainda não haviam sido projetados na época em que a aplicação foi criada.

O fato de a linguagem Lisp não ser orientada a objetos, limita sua atuação em um modelo baseado em objetos, como o OOXML. Grande parte das funcionalidades da orientação a objetos está diretamente associada à linguagem de programação. Por exemplo, a herança não se limita a um reaproveitamento de estruturas e processos entre classes ou objetos, mas pressupõe que a linguagem de programação dispõe de recursos capazes de explorar funcionalidades decorrentes da herança, tal como o polimorfismo.

Isto nos leva a crer que muitas funcionalidades proporcionadas por um modelo de objetos não poderão ser exploradas de forma adequada utilizando-se OOXML.

Coins

A proposta de Coins [ANP00] é centrar o projeto do sistema em torno da modelagem dos dados. Partindo do princípio de que XML é essencialmente voltado à representação de dados, a seqüência de concepção de um sistema baseado em Coins parte da elaboração da estrutura dos dados, seguido da produção do código a partir do mesmo.

Coins não pré-define uma relação fixa entre os elementos XML e as classes Java que implementarão o acesso e manipulação dos dados disponíveis nos elementos. Isto é feito a partir de esquemas intermediários que irão estabelecer esta ligação.

Tomemos um exemplo de um elemento XML com três atributos:

```
<Ficha nome=”Mila” idade=”21” sexo=”feminino”/>
```

Neste caso, queremos mapear este elemento de forma que represente um objeto da classe Ficha, com três atributos: nome, idade e sexo.

Documentos XML podem seguir um esquema de construção denominado DTD – *Document Type Definition* [BRA00].

A DTD deste elemento é a seguinte:

```
<!ELEMENT Ficha EMPTY>
<!ATTLIST Ficha
    nome  CDATA #REQUIRED
    idade CDATA #IMPLIED
    sexo  CDATA #IMPLIED>
```

Coins possui uma linguagem XML, denominada QDML, para a definição de documentos cuja função é equivalente à da DTD. Apresentamos a seguir a versão QDML[3] da DTD acima (a conversão pode ser feita automaticamente através de um utilitário):

```
<element name="Ficha" content="EMPTY">
    <attribute name="nome" status="REQUIRED"/>
    <attribute name="idade"/>
    <attribute name="sexo"/>
</element>
```

O arquivo QDML fornece suporte para a elaboração do arquivo QJML, que também é uma linguagem XML, com recursos adicionais para se estabelecer a maneira que cada elemento do arquivo original de dados XML será mapeado para classes em Java.

No exemplo a seguir, é apresentado um arquivo QJML que mapeia o elemento `Ficha` para a classe `business.data.Ficha` (o conteúdo do elemento `<targetClass>` estabelece a classe de mapeamento) e mapeia cada atributo XML para um atributo Java. Os nomes dos respectivos campos em Java são definidos por `field`.

```
<element name="Ficha" content="EMPTY">
    <targetClass>business.data.Ficha</targetClass>
    <attribute name="nome" status="REQUIRED" field="nome">
        <targetClass>java.lang.String</targetClass>
    </attribute>
    <attribute name="idade" field="idade">
        <targetClass>java.lang.Integer</targetClass>
    </attribute>
    <attribute name="sexo" field="sexo">
        <targetClass>java.lang.String</targetClass>
    </attribute>
</element>
```

A estrutura básica da classe equivalente a esta definição QJML deve ser um JavaBean e é definida como segue (o esqueleto desta classe pode ser criado automaticamente a partir de um utilitário):

[3] Os exemplos utilizados nesta explanação tomam como base a versão 3 do utilitário Quick [ANP00] que implementa Coins.

```
package business.data;
public class Ficha
{
      public String nome=null;
      public Integer idade=null;
      public String sexo=null;

      public Ficha()
      {
      };
}
```

Coins define um pacote de aplicações Java denominado Quick, que implementa utilitários para a realização das tarefas descritas, bem como uma API que permite a conversão de um documento XML em uma estrutura de objetos e vice-versa, a partir dos esquemas intermediários descritos.

Elementos Dinâmicos

Os documentos XML que são a origem dos dados contêm essencialmente valores de estado de objetos *Coin*. Uma vez estabelecidos os esquemas de conversão, a biblioteca Quick dispõe de funções que montam estruturas de objetos Java automaticamente, a partir da leitura de um documento XML e vice-versa.

Elementos Estáticos

Os métodos que definem o comportamento dos objetos não são serializados no documento XML em conjunto com os valores de estado, tal como faz o OOXML. Este comportamento está definido nas classes Java para as quais se realiza o mapeamento.

Esta estratégia também é adotada por outras propostas e certamente sofre influências do modo como as linguagens orientadas a objetos organizam seu código.

Nas linguagens orientadas a objetos, como Java, cada módulo de código, representado por um método, dificilmente pode ser destacado e utilizado fora do contexto de sua classe. Os métodos geralmente fazem uso de recursos diretamente ligados à classe, tais como: atributos, chamadas de outros métodos pertencentes à classe ou às superclasses, etc.. Por este motivo, serializar o comportamento do objeto envolve muito mais do que a simples serialização de métodos isolados, implica uma decisão entre serializar a classe por completo, ou apenas fazer referência à classe codificada em linguagem nativa.

Implementação

Apesar de dispor de uma filosofia de mapeamento XML ←→ Objetos, genérica o bastante para ser mapeada para a maioria das linguagens orientadas a objetos, Coins concentra toda a sua estrutura de implementação na linguagem Java.

O processo de conversão fica a cargo de programas projetados para trabalhar com Java. Adaptar Coins para outra linguagem significa não apenas redefinir os esquemas de mapeamento intermediários, como também adaptar todo o ferramental de conversão e tratamento dos documentos e objetos.

MONDO

MONDO constitui "uma arquitetura generalizada para codificação, modelagem e processamento de informações" [CHI97]. Para isto, ela integra informações de modelagem orientadas a objetos com descrições em linguagem de marcação XML e OML (*Object Markup Language*).

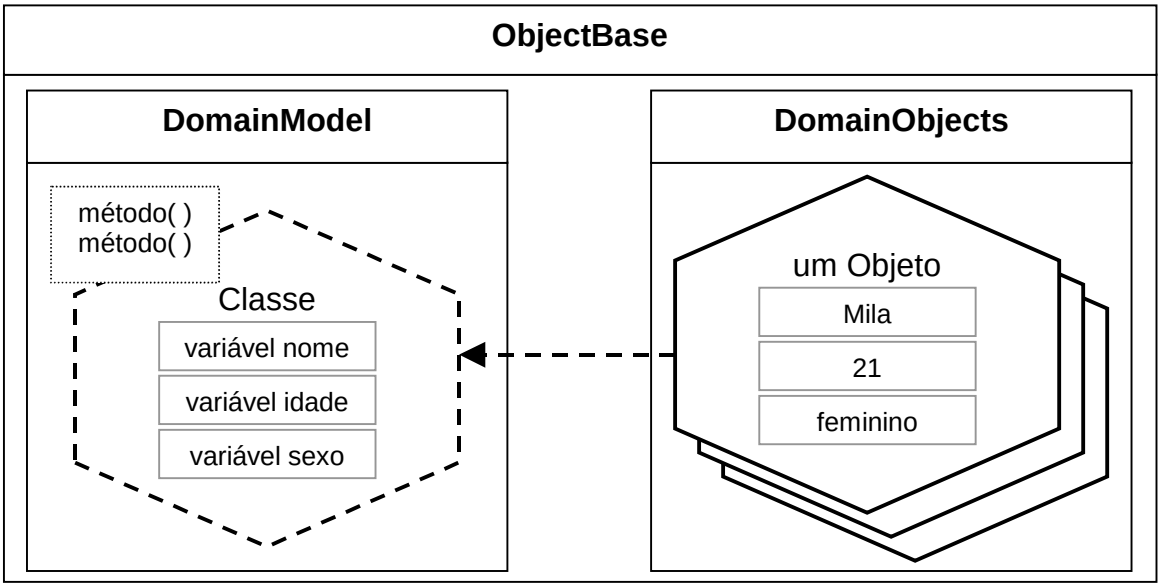


Figura 3-1: Diagrama ilustrando a inserção do DomainModel e do DomainObjects em uma ObjectBase.

Ao contrário das propostas anteriores que operam diretamente com arquivos XML, a arquitetura MONDO [CHI97] define uma base de objetos em formato próprio – ObjectBase – a partir da qual se opera todo o processamento. A representação dos objetos na base é dividida em duas partes (Figura 3-1):

- O DomainModel mantém todas as informações estáticas que dizem respeito aos objetos, tal como sua classe.

- O DomainObjects é responsável por manter as informações dinâmicas atuais, associadas à instância do objeto.

A fim de tornar a base plenamente acessível a outras aplicações, MONDO dispõe de dois subsistemas que realizam a conversão do formato interno para XML ou OML e vice-versa. Através do ObjectBuilder, dados de fontes externas (geralmente arquivos XML ou OML) são transformados em uma ObjectBase e, utilizando-se o ObjectEncoder, é possível se realizar o processo inverso.

Elementos Dinâmicos

O estado dos objetos fica armazenado na ObjectBase, especificamente no DomainObjects. As aplicações interagem diretamente com o subsistema, responsável pelo gerenciamento desta base, para obter acesso a estes objetos.

Elementos Estáticos

Também dentro da ObjectBase, especificamente no DomainModel, o sistema é capaz de armazenar o código binário de métodos de uma classe em uma linguagem de programação específica. Isto é especialmente interessante quando se trata de código móvel, tal como *bytecodes* em Java.

Este tipo de representação exige que a aplicação, que irá interagir com o subsistema da ObjectBase, suporte a execução do tipo de código armazenado.

Implementação

Conforme comenta Frank Manola [MAN98.2], o MONDO pode ser visto basicamente como um mecanismo para serialização de objetos. Deste modo, espera-se que uma aplicação externa interaja com a ObjectBase, que lhe oferecerá subsídios para o processamento dos mesmos, conforme sua necessidade.

A construção da ObjectBase pode ser feita a partir de um recurso XML ou OML. Para isto, MONDO utiliza um mecanismo intermediário, denominado Recipe (Figura 3-2). Recipes são “instruções para a construção de DomainObjects, que juntos irão representar conhecimento em um computador” [CHI97].

Através de um *parser*, um recurso XML – apropriadamente descrito conforme princípios estabelecidos por MONDO – é transformado em um Recipe. O Recipe atua como um modelo genérico, a partir do qual pode ser construído um objeto na ObjectBase, ou novamente codificado um recurso XML (ou OML).

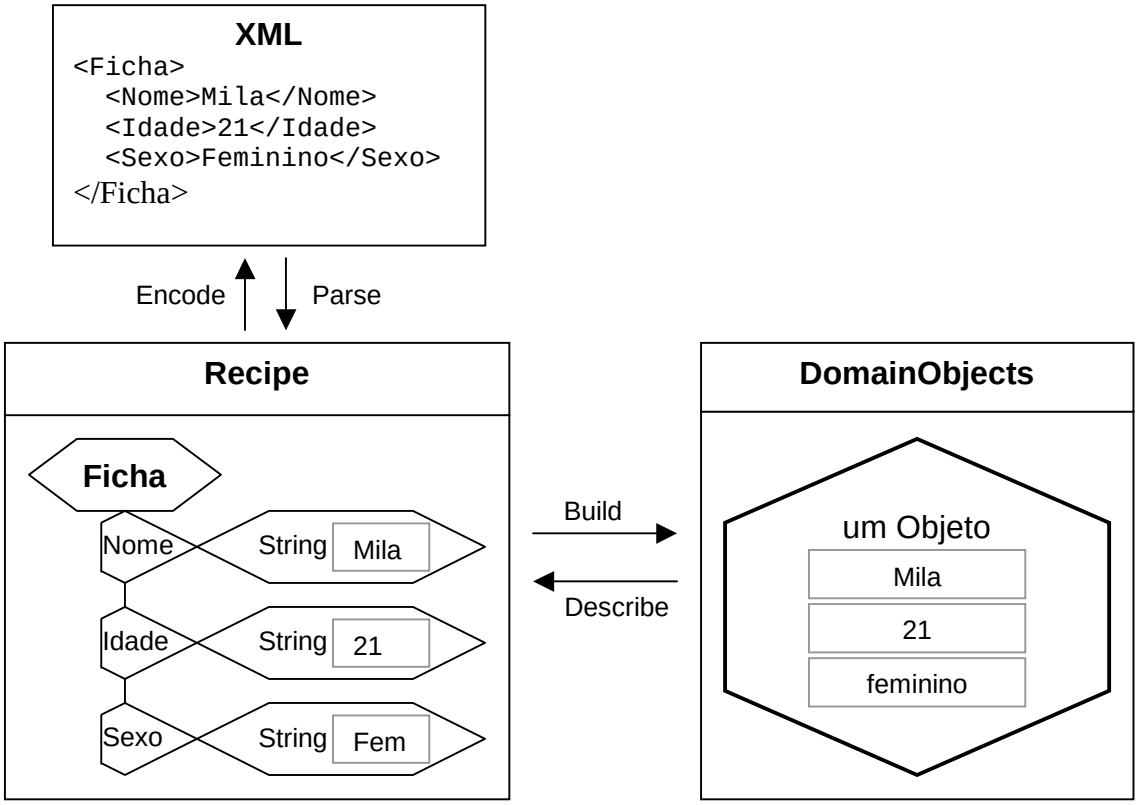


Figura 3-2: Diagrama geral de construção e decodificação de um DomainObject.

BML

O artigo de Mark Johnson [JOH99.1] apresenta um mecanismo para representação dos JavaBeans através da linguagem XML, de uma forma bem mais simplificada do que as apresentadas anteriormente e, por este motivo, menos genérica. Os *beans* são gerados a partir da leitura dos documentos XML, realizada por rotinas escritas em Java. Da mesma forma, a partir de *beans* disponíveis em uma aplicação, podem ser construídos documentos XML que os representem. Toda a proposta e mecanismos de conversão se baseiam especificamente em código escrito em Java.

Dentro deste mesmo princípio, a IBM concebeu BML – *Bean Markup Language* [JOH99.2], que expande o conceito, adicionando recursos para construção de aplicações completas.

A proposta da IBM se destaca por estabelecer um mecanismo claro de integração entre componentes, de modo que o documento BML não representa apenas cada objeto individualmente, como também o modo que eles se interligam para formar uma aplicação.

Elementos Dinâmicos

BML é uma linguagem XML e define marcadores que refletem a estrutura interna dos JavaBeans. Como pode ser observado no exemplo a seguir, o elemento <bean> define que será criado um *bean* cuja classe é `business.data.Ficha` e de identificação `Ficha`.

```
<bean class="business.data.Ficha" id=”Ficha”>
    <property name="nome" value="Mila"/>
    <property name="idade" value="21"/>
    <property name="sexo" value="Feminino"/>
</bean>
```

Cada elemento `<property>`, subordinado ao elemento `<bean>`, estabelece o valor de uma propriedade do componente.

Elementos Estáticos

A maior parte da funcionalidade dos objetos está implementada nos JavaBeans associados ao documento. No entanto, BML oferece alguns recursos que envolvem: ações na construção de uma aplicação, interligação entre objetos e *scripts*.

Entre os elementos que envolvem uma ação, por exemplo, está o `<add>`, que indica a inclusão de um componente em outro componente denominado *Container*. Esta é uma tarefa comum na definição de interfaces visuais, onde um componente contém outro.

O elemento `<event-binding>` estabelece ligação entre componentes. Ele reproduz o sistema de comunicação de componentes da linguagem Java baseado em eventos. Neste sistema, um componente A efetua um registro em um componente B, para escutar um evento X. Cada vez que este evento acontece, o componente B notifica o componente A, que executa alguma ação a respeito.

A ação pode ser descrita dentro do elemento `<event-binding>`, utilizando *scripts*, que seguem a linguagem BML, ou através de linguagens de *script* como o JavaScript.

Implementação

Existem duas formas de se executar uma aplicação definida em BML. A primeira é através do BML *player* (vide Figura 3-3) que realiza a leitura do documento e, através de um *parser*, o converte em uma árvore DOM [WOO98]. A partir desta árvore, o *player* cria a rede dos JavaBeans que irão compor a aplicação e a executa.

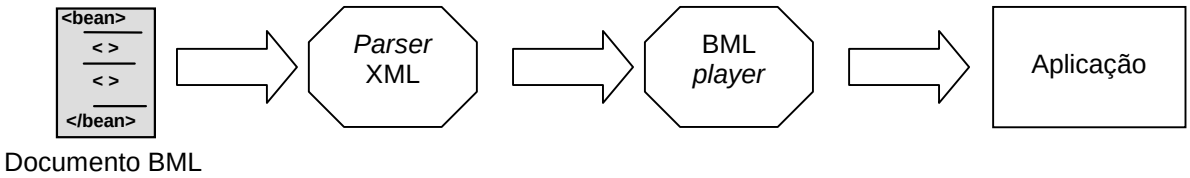


Figura 3-3: Diagrama de execução de uma aplicação através do BML *player*.

Quando o número de componentes é grande, este processo pode consumir uma quantidade memória elevada, além do tempo extra requerido para a análise e montagem dos componentes. Por ser um *player* genérico, capaz de operar com qualquer classe Java definida no documento BML, ele precisa utilizar a API Java *Reflection* [GRE98], que oferece recursos para a inspeção e o processamento de classes em tempo de execução. Isto também torna o processamento mais lento.

Por estes motivos, foi criada uma outra alternativa para a execução de uma aplicação através do BML *compiler* (vide Figura 3-4).

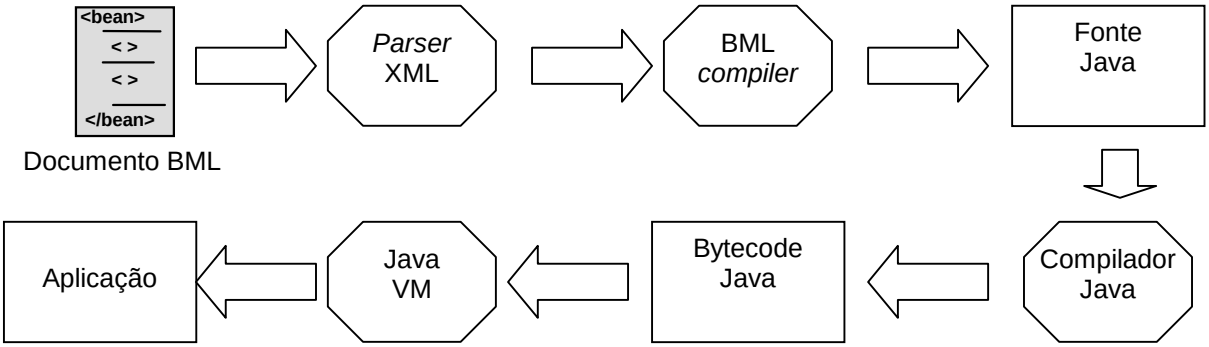


Figura 3-4: Diagrama de execução de uma aplicação através do BML *compiler*.

Neste caso, após a leitura e criação da árvore DOM, o BML *compiler* produz um código fonte Java, que representa toda a funcionalidade definida no documento BML. O código desta aplicação em Java tem menos exigências de memória e não necessita mais do Java *Reflection*.

***Long-Term Persistence for* JavaBeans**

Long-Term Persistence for JavaBeans [MIL99] consiste em um projeto da Sun Microsystems para a serialização de componentes e aplicações, utilizando XML. Ela parte de um recurso, já existente nos JavaBeans, que delega a cada componente a função de serializar seu conteúdo em um formato próprio, binário e compacto. A nova proposta consiste em adicionar em cada componente suporte à sua serialização em formato XML.

Elementos Dinâmicos

A serialização XML representa o estado dos componentes, as ligações entre eles e alguns comandos *script*.

A representação das instâncias dos componentes é feita através do elemento `<object>`. Seu conteúdo é convertido em uma seqüência de comandos responsáveis pela criação do objeto. No exemplo a seguir, está sendo definido um objeto da classe `business.data.Ficha`:

```
<object class="business.data.Ficha">
    <string>Mila</string>
    <int>21</int>
    <string>Feminino</string>
</object>
```

Os elementos subordinados representam os parâmetros que acompanham a chamada do construtor na mesma ordem. A tradução do elemento em Java é:

```
new business.data.Ficha(“Mila”, 21, “Feminino”);
```

Eventualmente, podem ser definidas propriedades que devem ser modificadas logo após a criação do objeto:

```
<object id=”ficha” class="business.data.Ficha">
    <void property="nome"><string>Mila</string></void>
    <void property=”idade”><int>21</int></void>
    <void property=”sexo”><string>Feminino</string></void>
</object>
```

Esta estrutura já se apresenta bastante semelhante à utilizada pelo BML. Em Java isto seria equivalente a:

```
ficha = new business.data.Ficha();
ficha.setNome(“Mila”);
ficha.setIdade(21);
ficha.setSexo(“Feminino”);
```

Elementos Estáticos

O comportamento dos componentes está implementado nas classes Java às quais eles pertencem. Ele não é serializado em XML juntamente com o estado do objeto. Através da classe do objeto, registrada na representação XML, é possível se definir qual classe implementa o comportamento do objeto.

O arquivo XML também pode conter alguns procedimentos, em especial aqueles relacionados com a interligação de componentes.

Este é o caso do elemento `<void method="método">` que corresponde à chamada de um método. Em Java, durante a construção da interface gráfica de um programa, é comum a anexação de um componente visual dentro de outro, através do método "`add`". Isto pode ser feito diretamente dentro do arquivo serializado, através do elemento `<void method="add">`.

Implementação

Para executar a codificação e decodificação dos *beans* em XML, são utilizadas duas classes fundamentais: `XMLEncoder` e `XMLDecoder`.

O `XMLEncoder` é responsável pela codificação do *bean* em formato XML. Ele possui um método – `writeObject` – que recebe como parâmetro a instância de um *bean* e o grava em uma *stream* de saída. Isto é possível quando o *bean* possui implementado seu método de serialização para XML, conforme descrito.

Para a leitura, instanciação e execução do conteúdo de um arquivo XML, basta utilizar o `XMLDecoder`.

JAXB

O Java *Architecture for* XML *Binding* (JAXB) [SUN01] faz parte do Java XML *Pack*, que reúne várias soluções da Sun Microsystems envolvendo a integração entre as linguagens Java e XML.

Long-Term Persistence for JavaBeans e JAXB estabelecem uma relação entre os objetos e XML em caminhos inversos. Enquanto o primeiro está focalizado nos componentes de *software* (JavaBeans) e utiliza XML para representá-los, o segundo parte dos dados XML a serem processados, estruturando um conjunto de classes que os representa e que serão utilizadas para sua manipulação. Sua perspectiva é equivalente à utilizada pelo Coins.

Tomemos com o exemplo o arquivo XML a seguir:

```
<Ficha>
    <Nome>Mila</Nome>
    <Idade>21</Idade>
    <Sexo>Feminino</Sexo>
</Ficha>
```

A DTD deste arquivo é definida como segue:

```
<!ELEMENT Ficha (Nome, Idade, Sexo) >
<!ELEMENT Nome (#PCDATA)>
<!ELEMENT Idade (#PCDATA)>
<!ELEMENT Sexo (#PCDATA)>
```

Para JAXB a DTD é um dos ingredientes que estabelece como será estruturada a classe Java. Além da DTD é necessário um esquema de associação – o *binding schema* – uma vez que ela não define claramente algumas características necessárias para a classe, tal como o tipo dos dados. O *binding schema* pode ser muito simples ou bastante detalhado. Quando ele é simples, o conversor estabelece algumas associações padronizadas. Tomemos, por exemplo, o seguinte esquema mínimo:

```
<xml-java-binding-schema>
    <element name="Ficha" type="class" root="true" />
</xml-java-binding-schema>
```

Ele estabelece que o elemento `Ficha` é a classe principal. O conversor irá deduzir que os elementos a ela subordinados representam atributos do tipo String.

A partir destes dois arquivos (DTD e *binding schema*) são criadas classes Java utilizando o compilador de *schema* (Figura 3-5).

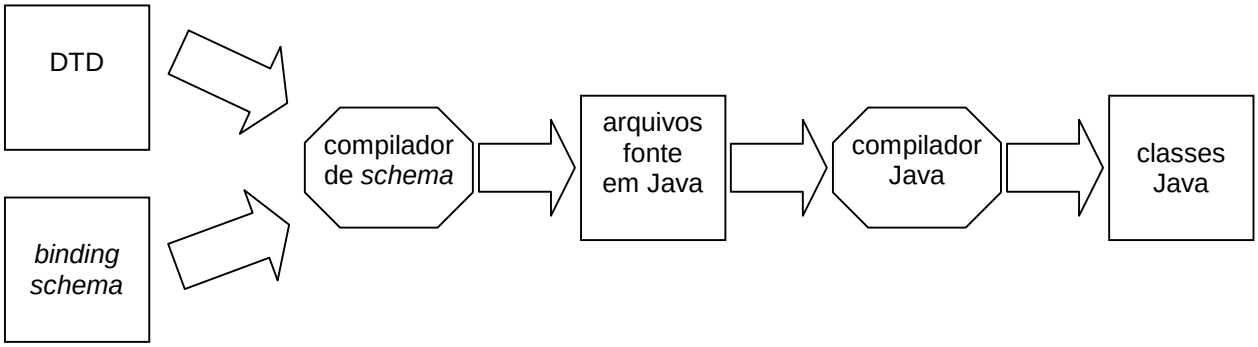


Figura 3-5: Diagrama de transformação da DTD em classes Java.

A classe resultante é definida com o seguinte construtor e propriedades:

```
public Ficha();
public String getNome();
public void setNome(String x);
public String getIdade();
public void setIdade(String x);
public String getSexo();
public void setSexo(String x);
```

Note que todas as propriedades estão definidas como String. A partir de um refinamento do *binding schema*, podem-se declarar tipos mais especializados, inclusive listas para elementos com mais de uma ocorrência.

Esta classe encapsula todo o acesso e manipulação dos dados XML em uma classe.

Elementos Dinâmicos

Tal como no *Coins*, os valores de estado dos objetos Java são derivados de dados do documento XML.

Elementos Estáticos

As classes produzidas pelo compilador de *schema* já dispõem do conjunto de métodos que serão necessários para as operações sobre os dados, tais como a conversão de um documento em objetos e vice-versa, acesso e modificação a valores, etc. Adicionalmente, podem-se acrescentar às mesmas novos métodos com propósitos especiais.

De qualquer modo, é importante ressaltar que o documento XML não define nenhum comportamento.

Implementação

A classe produzida pelo compilador dispõe de dois métodos responsáveis por realizar a ponte entre o XML e os objetos:

- **unmarshall**: carrega o documento XML e produz objetos da classe a ele associado, conforme foi produzido anteriormente pelo compilador de *schema*.
- **marshall**: deve ser chamado a partir de um objeto já existente, que irá traduzir seu conteúdo em XML.

Conforme aponta o guia do usuário JSB [SUN01], esta metodologia permite o tratamento de um arquivo XML, com esquema pré-estabelecido, através de um método mais rápido e que consome menos memória que os mecanismos genéricos, tal como o DOM.

3.2.1. Análise comparativa

Como pode ser observado, cada modelo realiza uma abordagem bastante diferenciada da integração entre objetos e XML. Nenhum deles, no entanto, possui todas as características exigidas pelo projeto *Anima*. Dentre as incompatibilidades e limitações encontradas reunimos as mais significativas, presentes em uma ou mais das propostas apresentadas:

- vinculação da representação dos objetos a uma linguagem de programação específica;
- anexação de código do comportamento à representação serializada do objeto;

- possui formato para serialização de objetos individuais, mas não possui formato adicional que represente a relação entre estes objetos para compor uma aplicação;
- representação muito limitada de aspectos estáticos dos objetos, como classe, interfaces e metadados.

Conseqüentemente, não foi possível adotar nenhuma das propostas na construção do modelo deste projeto. Por outro lado, a análise das vantagens de cada abordagem e suas limitações contribuiu na concepção do mesmo.

A seguir, apresentamos algumas considerações da análise realizada e abordamos, de forma mais detalhada, as incompatibilidades e limitações destacadas.

a) A representação dos objetos em XML pode ser dividida em dois grupos: os que produzem objetos a partir de um modelo XML e os que definem a estrutura XML para refletir o modelo de um objeto.

Para definir o esquema de marcadores utilizados nos documentos XML, os modelos analisados adotaram estratégias diferentes:

A primeira, utilizada pelo Coins e pelo JAXB, parte da lógica do documento. Isto significa que ao desenvolver uma aplicação, o responsável pela modelagem dos dados tem liberdade de estabelecer livremente as estruturas dos dados XML conforme sua necessidade, sem utilizar qualquer marcador pré-estabelecido. Um conjunto de esquemas intermediários estabelece como cada elemento e atributo do documento é mapeado para a estrutura interna dos objetos. Os objetos são criados para representar uma estrutura de dados.

Quando o objetivo é seguir o caminho inverso, ou seja, produzir estruturas XML que representem objetos modelados previamente em uma aplicação – como é o caso do MONDO, BML e *Long-Term Persistence for* JavaBeans – a abordagem é bem diferente. Os documentos XML possuem um conjunto padrão de marcadores baseados em uma DTD fixa (mas extensível). Através destes marcadores mapeia-se a estrutura interna dos objetos, definindo a classe do objeto, identificação, atributos, etc.

O OOXML adotou uma solução intermediária. Ele não estabelece marcadores pré-definidos para a maior parte da descrição, no entanto, interpreta os elementos XML sempre do mesmo modo – como uma hierarquia de objetos. Esta simplificação, não proporciona meios para uma descrição mais especializada (tipada) dos atributos do objeto.

Como pôde ser observado, cada abordagem é mais adequada para um domínio específico de trabalho. Isto justifica o fato de uma única empresa, como a Sun Microsystems, ter produzido dois produtos com abordagens diferentes.

b) Os modelos utilizam duas técnicas para a associação de comportamento aos objetos serializados: através da anexação de código diretamente na representação serializada do objeto ou através de uma referência à classe do objeto, onde está implementado o comportamento.

OOXML e MONDO optaram pela serialização de código dos métodos que descrevem o comportamento do objeto. Enquanto o OOXML coloca o código diretamente dentro da estrutura do objeto, seguindo a filosofia de modelos baseados em protótipos, o MONDO diferencia os aspectos estáticos do objeto, onde estão armazenados os métodos, configurando uma separação entre objeto e classe.

Independente deste fato, ambos utilizam o mesmo meio onde estão representados os dados da instância do objeto, para a representação dos métodos.

BML, *Long-Term Persistence for* JavaBeans, Coins e JAXB não representam em XML os aspectos estáticos do objeto, tal como a classe. Nos quatro casos, os métodos estão implementados na linguagem Java de forma independente da representação XML. Os dois primeiros fazem referência explícita à classe em seus objetos, já nos dois últimos esta referência é feita nos esquemas intermediários de transformação. Note-se que a opção pela linguagem Java por estes quatro modelos tem forte relação com as características desta linguagem, que por ser independente de plataforma, adequada à *Web* e pública, proporciona, em conjunto com XML, uma solução bastante genérica e portável.

Muitas são as implicações na escolha do local para inserção do código referente ao comportamento do objeto. Se por um lado anexar o código com a serialização do objeto oferece flexibilidade em termos da linguagem a ser utilizada, por outro, isto exige uma infra-estrutura em termos de: manutenção do código dentro dos objetos, implementação de interfaces entre o gerente dos objetos e as múltiplas linguagens ou criação de múltiplos gerentes de objetos em diversas linguagens, etc.

A disposição do código de comportamento do objeto em um recurso independente àquele da serialização exige uma infra-estrutura, para ligar o comportamento ao estado do objeto; por outro lado, o código da classe continua disposto conforme a estrutura nativa

definida pela linguagem ou pelo ambiente de desenvolvimento, facilitando a construção e manutenção do mesmo.

c) Padrões associados a uma linguagem de programação específica (Java, LISP, etc.) dificultam a integração de objetos em diferentes linguagens de programação.

De um modo geral, a maior parte dos padrões apresentados está associada a uma linguagem de programação específica.

O OOXML está associado ao Lisp e a aspectos específicos de seu funcionamento, tal como a capacidade de interpretação dinâmica de métodos. BML e as soluções da Sun estão diretamente ligadas à linguagem Java e a aspectos específicos de sua estrutura. Coins e MONDO apresentam estruturas de representação de dados bastante adequadas a um modelo genérico, independente de linguagem, no entanto, os mecanismos de associação de comportamento ao objeto ainda não são suficientemente genéricos.

No caso do Coins, toda a solução é apresentada em Java e, apesar de ser um modelo genérico, não parece viável criar um conjunto de ferramentas Quick para cada solução de linguagem adotada. O MONDO atua como uma base de objetos, que pode ter como clientes programas em diversas linguagens. Como não há mecanismos de integração, programas de cada linguagem ficam restritos a seu domínio de objetos, sem haver intercâmbio entre eles.

d) A maioria dos padrões se limita à serialização de objetos individuais, não abrangendo a dimensão de aplicação, que envolve outras questões tal como a representação da interligação entre os objetos.

Com exceção de BML e *Long-Term Persistence for* JavaBeans, os padrões oferecem suporte somente à serialização e recuperação de objetos individuais. Isto significa que seus mecanismos para gerenciamento de objetos sempre deverão estar acoplados a uma aplicação cliente, o que restringe estes modelos a um domínio específico de aplicações.

e) Os mecanismos que associam instâncias de objetos ao seu respectivo comportamento são definidos através de ferramentas, construídas para trabalhar com uma linguagem de programação específica.

Todas as propostas apresentadas utilizam uma ferramenta – *player* ou compilador – responsável pela associação de comportamento às instâncias dos objetos. Este mecanismo

limita o formato XML, aberto e independente de plataforma, a uma linguagem e implementação específicas.

Novas estruturas de associação, ou a modificação da linguagem de programação para a descrição do comportamento dos objetos, exigem a codificação de ferramentas específicas.

Conforme apresentaremos adiante, a linguagem XSL – *eXtensible Stylesheet Language* [ADL01], também uma linguagem XML aberta e independente de plataforma, oferece mecanismos de transformação capazes de associar as instâncias dos objetos a seu comportamento – através da conversão da representação genérica do objeto em implementações específicas – sem limitar o mecanismo de associação a uma determinada linguagem de programação.

f) A maior parte dos modelos se concentra na representação de aspectos dinâmicos do objeto, associados à sua instância, limitando a representação dos aspectos estáticos, tais como classe, interfaces e metadados.

Dentre os modelos analisados, apenas MONDO apresenta uma preocupação especial com aspectos estáticos associados ao objeto.

BML e *Long-Term Persistence for* JavaBeans representam essencialmente os aspectos dinâmicos do objeto e informações que permitem conectá-lo à respectiva classe Java. Coins e JAXB possuem informações para a conversão de esquemas XML em classes Java. Nestes casos, as informações disponíveis são suficientes para a execução das aplicações que envolvem os objetos. Por outro lado, quando se trata da criação e edição de aplicações, é desejável um conjunto bem maior de informações, que envolve uma descrição mais detalhada da classe, definição de interfaces e associação de metadados. Isto é especialmente importante quando a construção é baseada em componentes de *software*.

A tabela a seguir sintetiza os aspectos mais relevantes de uma análise comparativa, realizada entre os padrões apresentados:

	OOXML	Coins	MONDO	BML	LTP-JB	JAXB
a) Estratégia de representação Objeto – XML	objeto p/ XML	XML p/ objeto	objeto p/ XML	objeto p/ XML	objeto p/ XML	XML p/ objeto
b) Serialização do comportamento do objeto	×		×			
c) Vinculado à linguagem de programação	LISP	Java*	–	Java*	Java	Java
d) Abrangência da serialização	objetos	–	aplicação	aplicação	objetos	–
e) Mecanismo de associação em linguagem de programação.	×	×	×	×	×	×
f) Ênfase dada à representação de informações estáticas dos objetos	–	trans-formação	completa	conexão / trans-formação	conexão	trans-formação

* O padrão se propõe a ser genérico bastante para abranger outras linguagens, apesar de ser baseado na linguagem Java e estar fortemente vinculado à mesma.

Descrição de alguns indicadores utilizados na tabela			
a)	**Estratégia de representação Objeto – XML**	objeto p/ XML	Produz objetos a partir de um modelo XML.
		XML p/ objeto	Define a estrutura XML para refletir o modelo de um objeto.
d)	**Abrangência da serialização**	objetos	Limita à serialização de objetos individuais, não abrangendo a dimensão de aplicação.
		aplicação	Abrange a dimensão da aplicação.
f)	**Ênfase dada à representação de informações estáticas dos objetos**	transformação	Representa elementos estáticos do objeto suficientes apenas para realizar a transformação de esquemas XML em classes Java.
		conexão	Representa elementos estáticos do objeto, suficientes apenas para conectar objetos às respectivas classes.
		completa	Representação completa dos elementos estáticos do objeto.

No capítulo seguinte apresentamos o projeto *Anima*, seu modelo, estrutura e implementação. Boa parte das soluções adotadas leva em consideração alguns dos aspectos ressaltados neste capítulo, que serão confrontados com o modelo *Anima* nos próximos dois

capítulos. *Anima* se propõe a superar algumas das limitações destacadas no início deste tópico, através de um modelo que:

- não vincula a representação de objetos a uma linguagem de programação específica;
- estabelece uma separação entre a representação do estado dos objetos (independente de linguagem de programação) e a implementação de seu comportamento (codificada em linguagem de programação), definindo um mecanismo, independente de plataforma e implementação, que os vincula dinamicamente;
- define um modelo para representação de uma aplicação completa, envolvendo seus objetos e as interligações entre eles;
- define uma representação padronizada de informações referentes a aspectos estáticos dos objetos (classe, interface e metadados), independente de plataforma e implementação, que abrange diversos aspectos, geralmente não abordados por linguagens de programação.

4. ANIMA

Este capítulo apresenta os fundamentos que constituem o Projeto *Anima*. Ele está organizado em três tópicos: partindo de uma abordagem mais genérica, onde é apresentado o modelo abstrato que constitui a base de *Anima*, seguido de uma tradução do modelo para uma linguagem XML e concluindo com uma abordagem bastante específica, onde são apresentadas ferramentas que implementam o modelo proposto.

Seguindo esta seqüência, o primeiro tópico – **Modelo** – ordena a abordagem sobre quatro aspectos fundamentais do modelo: representação de uma aplicação constituída de objetos (composição); representação de classe, interface e metadados; protocolo de comunicação entre objetos baseado em mensagens e, finalmente, o mecanismo de transformação de uma representação abstrata em uma aplicação. Para cada aspecto é apresentado um modelo abstrato, não associado a nenhum tipo de linguagem, implementação ou plataforma. O modelo é concebido de tal maneira que possa ser mapeado para múltiplas linguagens.

O tópico subseqüente – **Representação baseada em marcação estruturada** – realiza um mapeamento do modelo genérico do tópico anterior para a linguagem XML e linguagens derivadas. É definida uma linguagem XML para representação do estado dos objetos e sua interligação na constituição de uma aplicação. Para representar a classe, interface e metadados, utiliza-se uma *framework* estruturada em XML, apropriada para a definição de metadados, denominada RDF. O protocolo de comunicação entre os objetos também se processa através de uma linguagem XML e o mecanismo de transformação de uma representação abstrata em uma aplicação é implementado através de outra linguagem, derivada de XML e específica para este propósito, denominada XSL.

Concluímos com o tópico **Ferramentas**, que apresenta um conjunto de ferramentas implementadas em Java, conforme o modelo *Anima*, e sua respectiva representação em XML. Inicialmente o tópico apresenta um conjunto de bibliotecas organizadas e documentadas de tal modo que possam servir como referencial de implementação do modelo. Estas bibliotecas também constituem uma camada que permite o fácil acesso de qualquer aplicação que pretenda utilizar este modelo. Em seguida, são apresentadas duas aplicações construídas para dar suporte a projetos que utilizem o modelo *Anima*. A primeira permite a captura, edição e armazenamento de arquivos contendo definição de classes, interfaces e metadados. A segunda

consiste de uma ferramenta de autoria que permite a criação e edição de aplicações a partir da combinação de componentes de *software* que seguem o padrão *Anima*.

4.1. Estrutura do Projeto

O Projeto *Anima* foi impulsionado por um conjunto de necessidades constatadas durante a elaboração de ferramentas para o desenvolvimento de atividades de ensino-aprendizagem fazendo uso do computador, em especial o Sistema Casa Mágica [SAN00].

O modelo de trabalho do sistema Casa Mágica, pautado sobre objetos, e as possibilidades decorrentes desta concepção, foram os principais elementos que deram origem às seguintes demandas:

- A construção de projetos em diferentes áreas exige diversos tipos de componentes, com qualidades e concepções associadas ao seu domínio. Em muitos casos, outros sistemas externos ao ambiente Casa Mágica, e eventualmente escritos em outras linguagens, possuem tais componentes, ou ferramentas mais apropriadas para a sua construção.
- O advento de padrões de marcação extensíveis, tal como XML, e a difusão de linguagens e sistemas capazes de executar aplicações em múltiplas plataformas e também em navegadores *Web* multiplicaram a possibilidade de intercâmbio de produções. Em termos de educação, isto abre a possibilidade para uma ampla colaboração. Para isto, torna-se necessário um padrão que possibilite interligar diversas produções independentes de forma simples e transparente.

O princípio sobre o qual está estruturado o modelo de *Anima* fundamenta-se em um deslocamento de dados, utilizados por sistemas e linguagens baseados em objetos – que originalmente são representados em um formato particular – para uma camada neutra, independente de plataforma e implementação.

Deste modo, o primeiro passo é estabelecer que dados pretendemos representar nessa camada e quais podem ser efetivamente deslocados. Isto não significa apenas possibilitar a comunicação entre objetos, independente da linguagem em que foram implementados, mas envolve a representação de todos os dados que impliquem a produção e execução de aplicações deste tipo. Ao pensarmos em criar meios para que objetos se integrem, independente de sua linguagem ou implementação, é preciso que pensemos também nas ferramentas que serão capazes de integrá-los e de produzir aplicações.

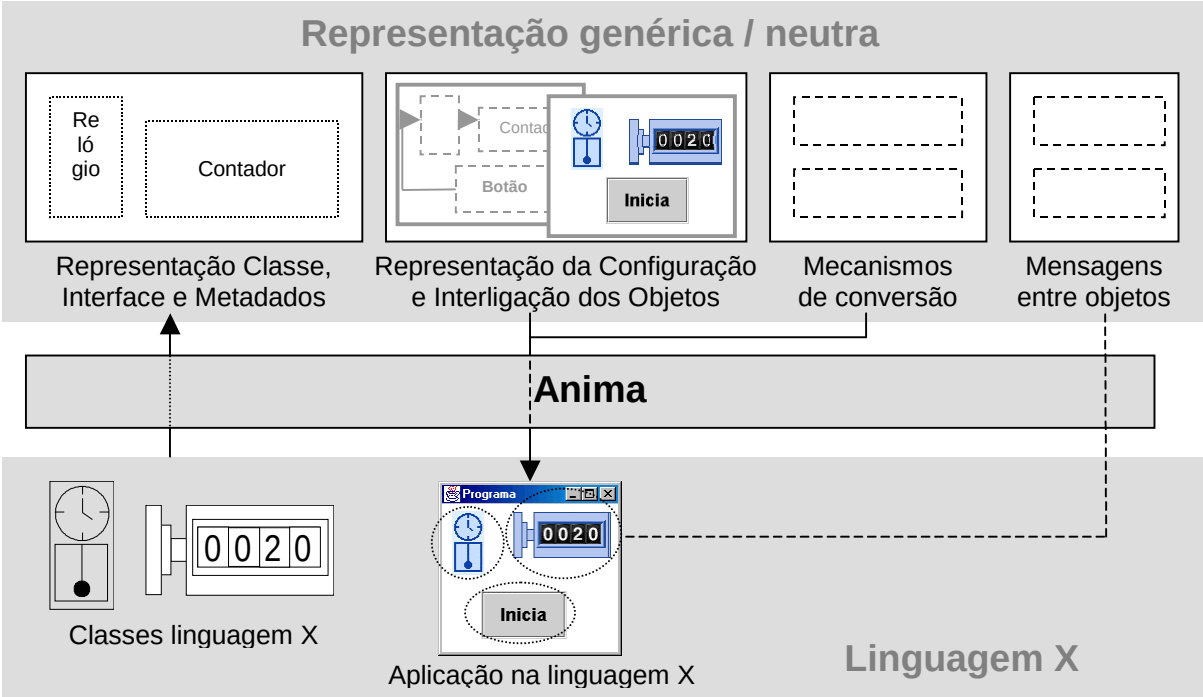


Figura 4-1: Deslocamento de representações para uma camada neutra.

Do ponto de vista da produção de aplicações, é necessário obter informações referentes à classe dos objetos que serão utilizados de uma forma homogênea. Isto implica o deslocamento das representações de classe de formatos específicos para uma representação neutra, o que propicia, inclusive, o enriquecimento desta representação com informações de interface e metadados descritivos, que geralmente não estão disponíveis nos formatos utilizados por linguagens de programação.

Construir aplicações que envolvem diversos objetos, independentes de sua linguagem, exige também um formato neutro que permita descrever como os objetos se configuram e como se interligam em cada aplicação diferente. Conseqüentemente, é interessante estabelecer que mecanismos serão utilizados para permitir que a representação neutra se converta em uma aplicação codificada em uma linguagem (ou em um conjunto de linguagens) de programação específica, de modo que possa ser executada.

Finalmente, é necessário, também, estabelecer um formato neutro para a troca de mensagens entre objetos, de modo que possam se comunicar, independentes da linguagem em que foram implementados.

O diagrama da Figura 4-1 sintetiza o deslocamento explanado.

A seguir, apresentamos o quadro referencial geral de todo o projeto *Anima* (Figura 4-2), com seu modelo de representação e sua relação com outras ferramentas:

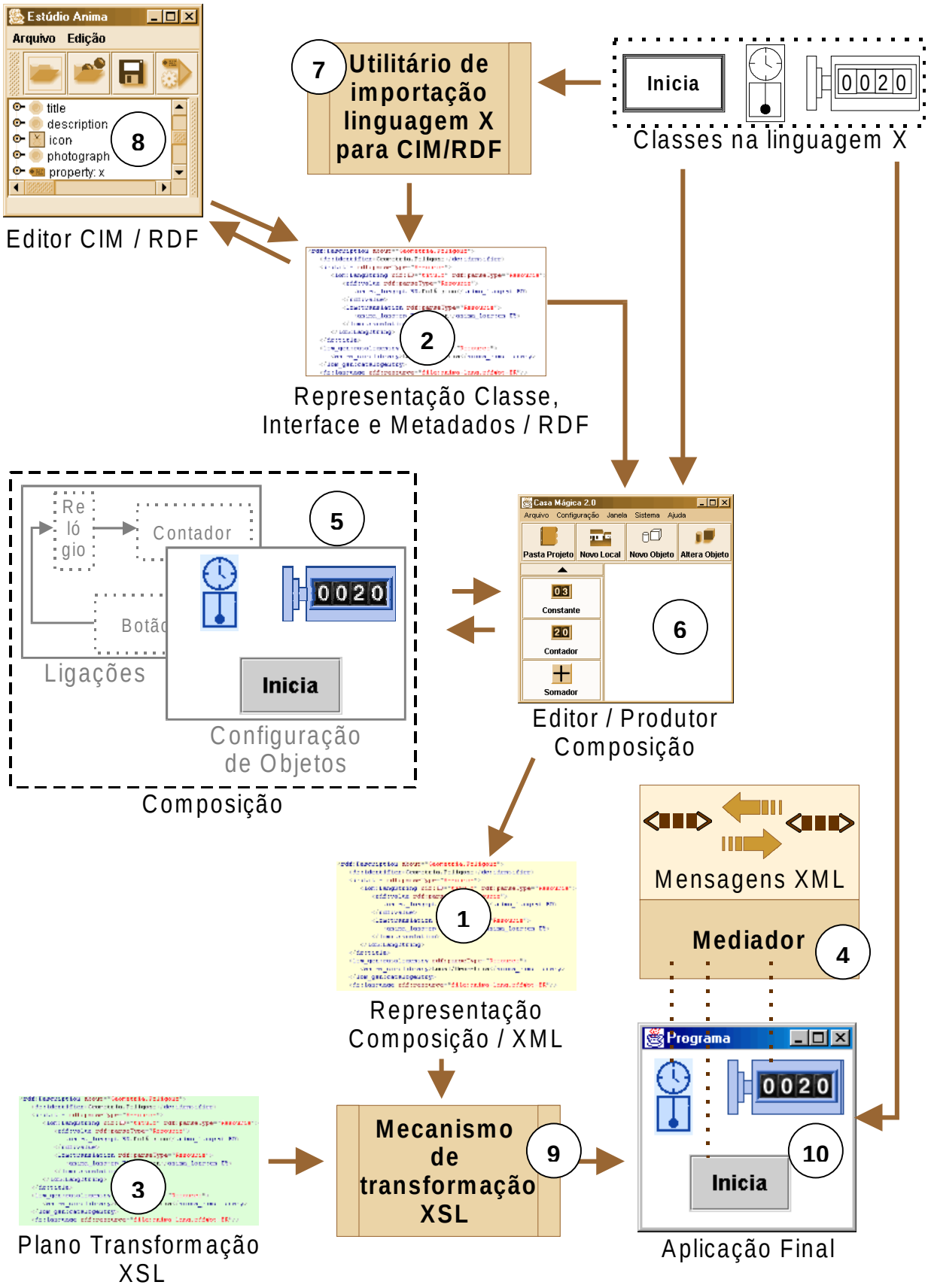


Figura 4-2: Diagrama geral do projeto *Anima*.

Apesar de surgir a partir do sistema Casa Mágica, o propósito do projeto *Anima* é justamente ampliar os horizontes para outros sistemas e linguagens.

O projeto *Anima* abrange os seguintes aspectos:

- **Modelo genérico**, independente de plataforma e linguagem para a representação de objetos e composições[1], de modo que possam ser distribuídos através da *Web*. Ele se subdivide em:
  - **Representação da composição** (Figura 4-2, indicador ①) – define um formato para representação de objetos, bastante genérico para ser utilizado por diversas linguagens e sistemas baseados em objetos. Este formato é utilizado para a representação de composições e para a transferência de objetos pela rede.
  - **Representação de classe, interface e metadados** (Figura 4-2, indicador ②) – define um formato aberto para representação de informações relacionadas ao objeto, tais como: sua classe, interface, eventos a que responde, metadados (como os definidos pelo LOM - *Learning Object Metadata* [HOD02]), etc.
- **Mecanismos de conversão** (Figura 4-2, indicador ③) padronizados que realizam a transposição dos objetos e composições da representação genérica para implementações específicas.
- **Protocolo padronizado de comunicação** (Figura 4-2, indicador ④) entre componentes dentro da composição, inclusive em ambientes mistos (o conceito de composições mistas será explorado adiante).

O modelo e os mecanismos implementados possuem características suficientemente genéricas para serem aplicadas em outros contextos, além do educacional. Por este motivo, eles podem servir de contribuição para um estudo mais abrangente em áreas que se aplicam à orientação a objetos, tal como uma representação genérica de objetos na *Web*.

Inicialmente, detalharemos cada um dos aspectos do projeto, na forma de um modelo abstrato sobre o qual se fundamentarão as demais etapas.

---

[1] O termo "composição" será utilizado nos próximos capítulos para a representação de um subconjunto de aplicações formadas de objetos, conforme estrutura definida por *Anima*. O tema será tratado em mais detalhes adiante.

4.2. Modelo

Antes da estruturação dos aspectos de representação XML e implementação de *Anima*, foi definido um modelo abstrato descritivo, a partir do qual pode-se derivar para diferentes formatos de representação e modalidades de implementação.

Esta estratégia reflete a seqüência de concepção do projeto (modelo abstrato → representação XML → implementação Java) e conduz à visão mais abrangente de *Anima*. A representação XML não é apresentada como uma restrição do modelo mais genérico, mas como uma possível representação, bastante adequada ao propósito do projeto.

O modelo é apresentado de forma puramente descritiva e, quando necessário, convencionamos algum tipo de notação para a apresentação de variações de um recurso, ou montamos esquemas que especificam a estrutura de uma representação. Sua apresentação está dividida conforme cada um dos aspectos apresentados anteriormente.

4.2.1. Representação da Composição

O universo do *software* educacional abrange um conjunto enorme de categorias, onde podemos encontrar programas de computador que seguem estruturas diferentes de implementação. A maneira mais flexível e abrangente de desenvolver estes produtos continua sendo a sua codificação em uma linguagem de programação.

Desenvolver aplicações educacionais constitui a principal meta de *Anima* que, para isso, adotou uma estratégia particular. Esta estratégia define um mecanismo para a combinação de objetos na formação de uma aplicação que alia simplicidade e versatilidade. Apesar de possuir restrições de flexibilidade, quando comparado ao uso de linguagens de programação, este mecanismo é mais simples. Conforme iremos aprofundar a seguir, a opção pela simplicidade torna *Anima* mais adequado para sua aplicação em atividades de ensino-aprendizagem e, em contrapartida, suas restrições de flexibilidade não afetarão a maior parte dos projetos em que ele esteja envolvido, principalmente quando é realizado um planejamento apropriado, conforme demonstraremos no capítulo 5.

A popularização dos componentes de *software* se deu em conjunto com o surgimento de ferramentas visuais, que os utilizava para o desenvolvimento de aplicações, oferecendo recursos para que o programador adicione e configure os componentes por manipulação direta através da interface visual.

Para que os componentes individuais alcancem a dimensão de aplicação, eles precisam interagir. A maioria das ferramentas oferece suporte à interação através da inserção de código,

associado ao componente, que é disparado ao ocorrer algum tipo de evento com esse componente. Quando a linguagem é orientada a objetos, o código fica disposto dentro de um método. A Figura 4-3 ilustra este procedimento, implementado através de uma ferramenta de programação visual para Java.

Os programas são divididos em porções associadas a eventos – programação orientada a eventos. Este tipo de programação é bastante flexível e muito adequada para os sistemas modernos, que interagem com o usuário através de interfaces visuais, apesar de não estar restrita a esta modalidade de aplicação.

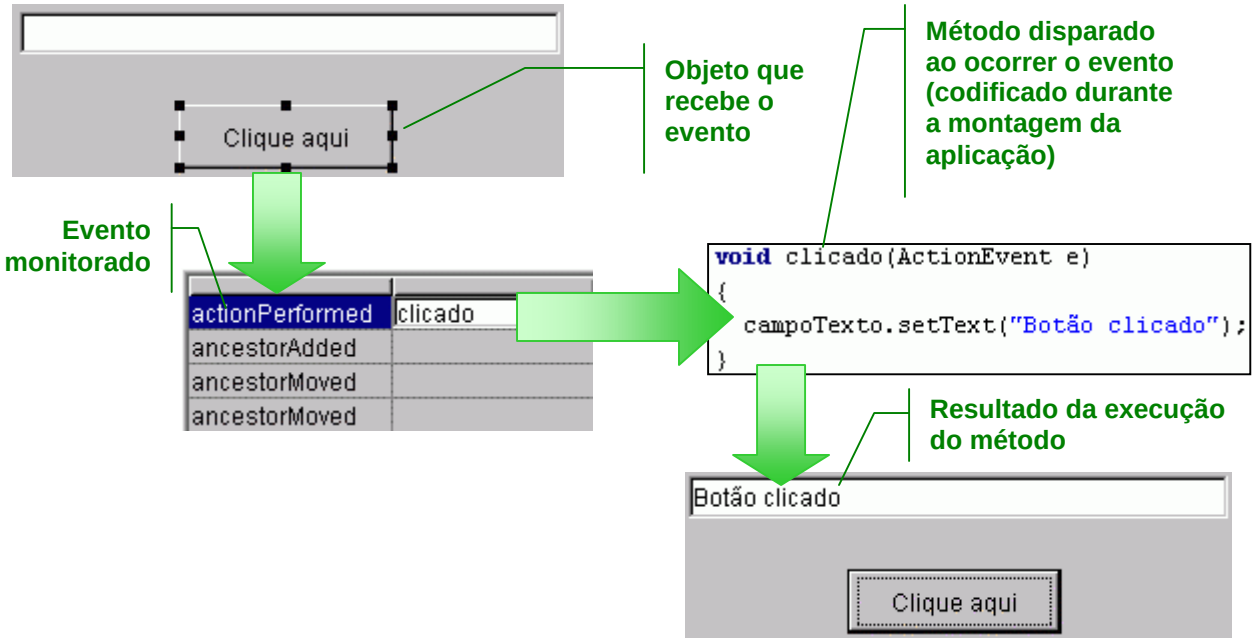


Figura 4-3: Diagrama de associação de código a objetos, a partir de um evento, em Java.

Apesar da flexibilidade, existem duas restrições observadas:

- A necessidade de inserção de código exige que o autor da aplicação conheça a linguagem de programação envolvida. Isto dificulta o desenvolvimento de aplicações por usuários não especializados em informática, como professores em geral e alunos.

- Inserir código associado a componentes para realizar sua interação exige que este código seja serializado em conjunto com a coleção de objetos, na representação de uma aplicação completa, o que cria uma dependência desta representação com uma linguagem de programação específica.

Existe outra estratégia para se interligar componentes que é bem mais simples e dispensa a inserção de código para a integração entre objetos. Para interligar dois objetos, esta

estratégia registra apenas um conjunto básico de informações, que realizam a associação objeto_origem/evento→objeto_destino/método, ou seja, ao ocorrer um evento em um objeto (origem), este envia uma mensagem – que resulta no acionamento de um método – a um outro objeto (destino). Apenas os dados-chave para a execução desta associação são armazenados, sem a inserção de código em linguagem de programação.

Apesar da simplicidade desta estratégia, o autor da aplicação estará limitado aos tipos de interação previstos nas classes dos componentes envolvidos no projeto.

Esta estratégia é utilizada em ambientes como o BeanBox [HAM97] da Sun Microsystems e o sistema Casa Mágica. A Figura 4-4 ilustra este tipo de associação no BeanBox.

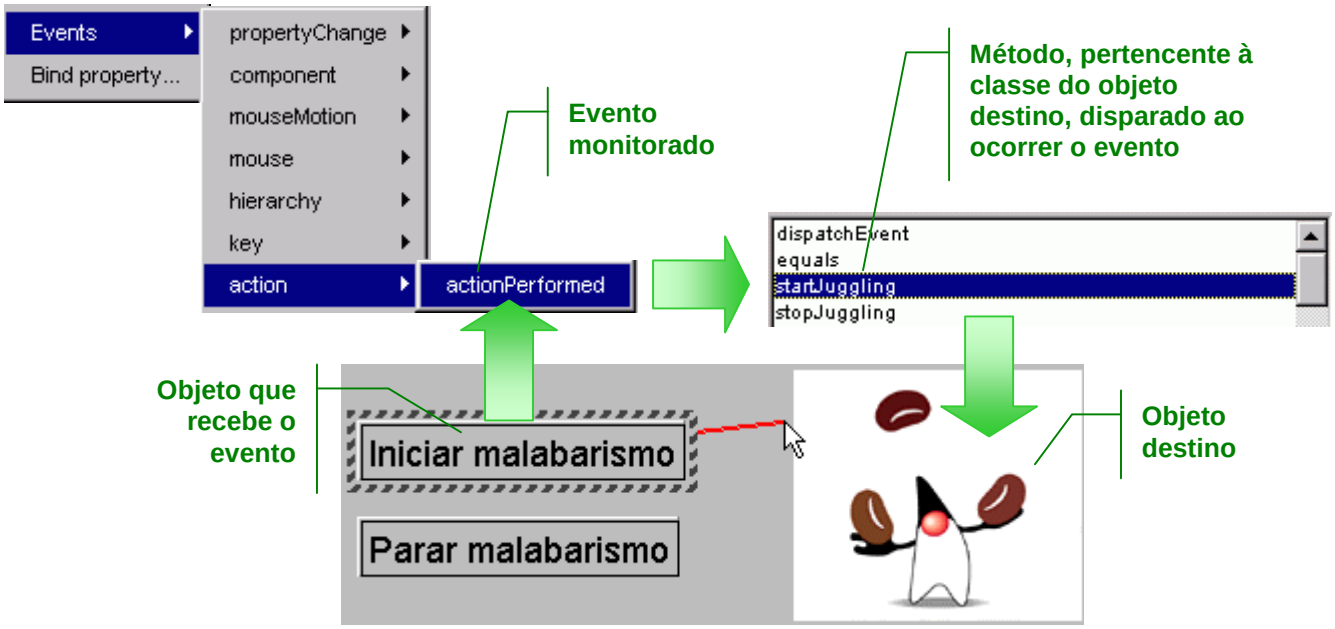


Figura 4-4: Diagrama de associação de objetos, a partir de um evento, no BeanBox.

Por trás da associação o BeanBox produz uma classe, cujo objeto realiza a função de constantemente verificar se o evento ocorreu no objeto de origem, ele é um *event listener*. Ao verificar a ocorrência do evento ele dispara o método do objeto destino.

O sistema Casa Mágica mapeia esta associação de modo diferente. Ele registra no objeto de origem as informações: evento, identificação do objeto destino e mensagem. Isto evita a criação de classes ou métodos para a associação, tratando-a como um conjunto de dados facilmente serializável.

Como já foi mencionado, esta modalidade para a construção de aplicações limita a interação dos componentes a categorias previstas na codificação de suas classes, pelo fato de

apenas registrar informações de associação entre objetos, sem a inserção de qualquer tipo de código de programação, o que permitiria o uso de rotinas de código para a criação de novas modalidades de interação entre os objetos, além daquelas previstas, durante a associação dos mesmos. Isto torna o modelo menos flexível em relação àquele que estabelece a inserção de código associado a eventos, especialmente quando o autor da aplicação necessita realizar algum tipo de operação não prevista pelo código do componente, tais como conversões, verificações, etc..

Anima optou pela solução usada pelo sistema Casa Mágica, tornando o desenvolvimento das aplicações mais acessível a usuários não especialistas em informática e evitando a inserção de código diretamente na representação serializada da aplicação.

Acreditamos que os benefícios decorrentes desta estratégia justificam sua escolha, apesar de suas limitações. Além disso, no que diz respeito à sua aplicação na educação, em grande parte das situações estas limitações podem ser contornadas, sem que isto resulte em prejuízo para o usuário. Destacamos algumas destas situações:

- Em atividades de ensino-aprendizagem preparadas para uso em sala de aula, onde os alunos irão combinar os objetos, pode-se estabelecer um conjunto bastante abrangente e versátil de componentes, que atendam aos objetivos de cada projeto, sem que seja necessário que o aluno-autor inclua qualquer código extra, enquanto realiza a atividade envolvendo a interligação dos componentes. Isso é viável devido à possibilidade de restrição do domínio, sobre o qual cada projeto se desenvolverá.

- Existem muitas operações freqüentemente codificadas em rotinas, associadas a eventos de componentes, que podem ser implementadas de outra maneira, através de um componente exclusivamente projetado para executar a operação. Eventualmente, por exemplo, um componente A precisa realizar a conversão de um valor antes de enviá-lo a outro componente B. Isso pode ser feito através de um componente conversor especializado, que recebe o valor de A, realiza a conversão e o envia para B.

- Equipes para a produção de aplicações podem dividir suas tarefas conforme sua especialidade, deste modo, professores e especialistas no domínio para o qual é destinada a aplicação, desenvolvem o projeto e selecionam os componentes envolvidos na produção. Programadores especializados codificam os componentes

adicionais não disponíveis e, quando necessário, adaptam os componentes disponíveis. Isso permite que componentes sejam feitos sob medida para cada projeto e que os autores (professores ou especialistas no domínio da aplicação) possam construir aplicações interligando componentes, sem a necessidade de inclusão de qualquer código extra.

Adotamos o termo "composição" para indicar este tipo de aplicação específica (Figura 4-2, indicador ⑤), elaborada a partir de uma simples interligação entre objetos, utilizando associações do tipo objeto_origem/evento→objeto_destino/método.

A representação de composições em *Anima* envolve três aspectos:

- Caracterização do espaço de trabalho onde será montada a composição.
- Serialização dos objetos envolvidos na composição.
- Interligação entre objetos que realizarão troca de mensagens.

Vamos tratar cada um destes aspectos separadamente.

4.2.1.1. Espaço de trabalho

No nível hierárquico mais alto do modelo, está o espaço de trabalho (*workspace*), que estabelece uma área física na qual os objetos irão atuar. É possível estabelecer uma hierarquia de espaços de trabalho, de modo que, dentro de um espaço mais abrangente (raiz) são dispostos outros espaços subordinados (vide Figura 4-5), conforme a necessidade de divisão física. Isto será especialmente importante quando a composição envolver objetos de diferentes linguagens de programação e não for possível o compartilhamento da mesma área física por dois objetos com implementações diversas.

O espaço de trabalho de mais alto nível determina quais os ambientes em que se pode executar a composição (ex.: navegador *Web*, aplicação Java, etc.), e qual deles é o preferencial. Cada espaço subordinado sempre tem que ser compatível com o espaço do nível acima. Isso significa que a aplicação resultante, produzida a partir do espaço de nível mais alto, tem que ser capaz de incluir e executar, dentro de seu espaço físico, uma aplicação resultante do espaço de trabalho subordinado. No exemplo, apresentado na Figura 4-5, os espaços de trabalho subordinados são compatíveis com o espaço de trabalho de nível superior (HTML), pois tanto Casa Mágica quanto Alice oferecem recursos para inclusão e execução de suas aplicações dentro de uma página HTML.

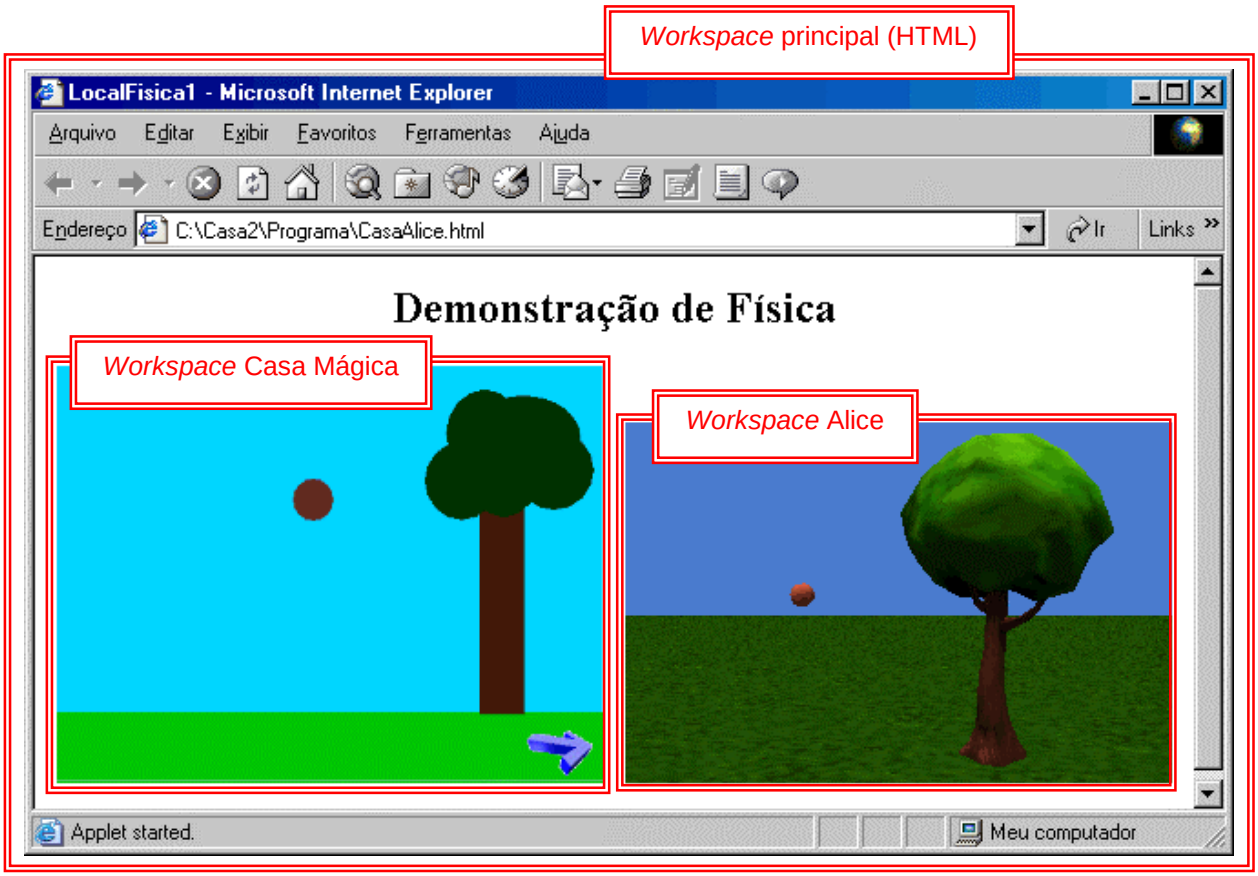


Figura 4-5: Divisão de espaços de trabalho (*workspaces*) entre objetos escritos em linguagens diferentes.

A verificação de compatibilidade entre espaços de trabalho é feita através de um registro independente, onde existe uma 'lista de compatibilidade' associada a cada linguagem. Esta lista contém uma relação de outras linguagens compatíveis quando associadas a espaços de trabalho de nível mais alto.

Ex: (Casa Mágica: Java, HTML)
(Alice: HTML, Squeak)

Estes espaços determinam o modo como o objeto se apresentará na composição resultante. Por este motivo, como analisaremos mais adiante, os mecanismos de conversão irão adaptar o resultado conforme informações contextuais referentes ao espaço de trabalho.

Espaços de trabalho podem ser tratados como uma modalidade especial de objeto, que envolve todos os demais objetos. Do ponto de vista visual eles se comportam como um *container*. Suas principais funções são: estabelecer o ambiente/linguagem destino para conversão do espaço de trabalho, delimitar o espaço físico onde os objetos são apresentados e definir mapeamentos entre o ambiente interno e externo.

A inserção de um espaço de trabalho dentro de outro se processa de modo diferente, de acordo com a linguagem utilizada. Em HTML, por exemplo, existem marcadores específicos para inserção de objetos estrangeiros, tais como: `<applet>`, `<object>` ou `<embed>`. Estes objetos estrangeiros podem ser programas em linguagens de programação, gerados a partir dos espaços de trabalho subordinados à página.

Para a representação de um espaço de trabalho, estabelecemos um esquema, denominado *Workspace Schema* (Apêndice B.1). Sua função é detalhar cada elemento envolvido na descrição de um espaço de trabalho, em um nível de abstração que não envolve qualquer linguagem ou implementação específica.

Por este motivo, foram utilizadas tabelas que representam os esquemas ou sub-esquemas. Estes últimos descrevem elementos cujos tipos são coleções de outros elementos.

Na medida do possível, foram detalhados os formatos e tipos associados ao conteúdo de cada elemento. Quando necessário, referenciamos padrões utilizados na representação de certas categorias de informações.

Uma composição é formada, então, por um espaço de trabalho de nível hierárquico mais alto e eventualmente por espaços de trabalho subordinados. Dentro dos espaços de trabalho, estão inseridos os objetos e suas ligações, organizados em duas camadas distintas, conforme pode ser observado na Figura 4-6.

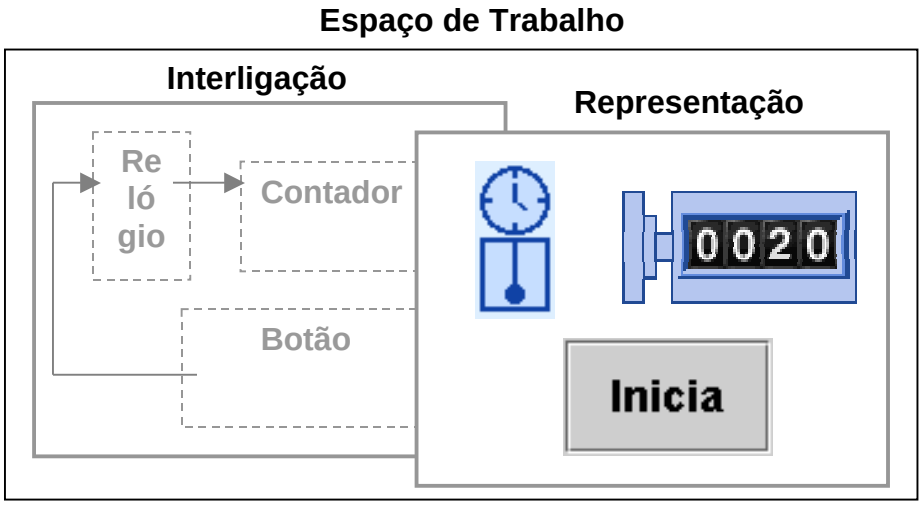


Figura 4-6: Representação de objetos e sua interligação em um espaço de trabalho.

Na camada de representação, estão dispostas as configurações individuais de cada objeto. A camada de Interligação define o modo como estes objetos estão ligados, conforme esquema objeto_origem/evento→objeto_destino/método, já mencionado.

4.2.1.2. Representação do Objeto

Os objetos ficam dispostos dentro dos espaços de trabalho. Desta maneira cada espaço tem um ou mais objetos, que poderão interagir uns com os outros.

Por ser um modelo baseado em classes, Anima realiza a separação classe-objeto. Objetos da mesma classe possuem estrutura e métodos iguais. Eles se diferenciam por sua identidade e podem ter valores distintos para seus atributos (Figura 4-7).

Classe Botão				
Id: X	**Atributos**		**Id: Y**	**Atributos**
Liga	**Rótulo:** Liga **Largura:** 60 **Altura:** 40		Desliga	**Rótulo:** Desliga **Largura:** 50 **Altura:** 20

Figura 4-7: Dois objetos da mesma classe, com identidade e valor dos atributos diferentes.

Anima separa a representação da classe em um recurso independente, onde ficam as características estáticas do objeto: classe, interface e metadados (CIM). Esta representação será tratada mais adiante.

A representação dos objetos na composição estabelece: a classe a que cada um pertence, a fim de associá-los ao respectivo recurso CIM, sua identidade e valor dos atributos. Isto é definido em Anima conforme o *Object Schema* (Apêndice B.2).

O *Object Schema* estabelece um identificador do objeto, que é utilizado em princípio para uma associação local entre objetos. Em geral, cada vez que o objeto é montado na memória, pelo sistema ou linguagem alvo, ele recebe uma identidade (geralmente seu endereço na memória ou algo equivalente [SZP99]). Esta identidade muda a cada nova execução. Por este motivo, este identificador pode ser considerado uma referência, que se associa à identidade do objeto em tempo de execução [SZP99].

Os atributos podem ser do tipo simples ou pertencer a alguma classe; no segundo caso seu valor será a representação do objeto desta classe, conforme o *Object Schema*.

4.2.1.3. Interligação entre objetos

Já abordamos anteriormente o método particular de interligação entre objetos, utilizado por *Anima*, que se baseia no registro de informações que associam objetos. Esta interligação é definida pelas seguintes informações (ilustrado na Figura 4-8):

- **Objeto de origem** sobre o qual ocorre o evento. Ele é responsável por enviar uma mensagem para o objeto destino, que irá disparar a execução de um método. Em *Anima*, o registro completo da interligação fica dentro do objeto de origem.
- **Evento** associado ao objeto de origem que dispara todo o processo. O evento pode ser um clicar de *mouse*, quando o cursor está sobre o objeto, pode ser uma mensagem recebida de outro objeto, etc.. Todo evento é representado em *Anima* na forma de mensagens, que são enviadas por outros objetos ou pelo sistema para o objeto que recebe o evento.
- **Mensagem** que será enviada para o objeto destino quando acontecer o evento. O formato desta mensagem pode ser simples, uma *string*, por exemplo, ou pode ser composta. As mensagens serão discutidas em detalhes mais adiante.
- **Esquema da mensagem** enviada para o objeto de destino. Cada objeto é capaz de enviar diferentes tipos de mensagem para seu destino, que podem variar conforme o evento. O esquema determina qual o tipo específico de mensagem desejada. Esquemas para mensagens serão discutidos em detalhes mais adiante.
- **Objeto alvo** que recebe a mensagem. A referência ao objeto alvo é feita através do seu identificador no espaço de trabalho, ou em espaços de trabalho vinculados.

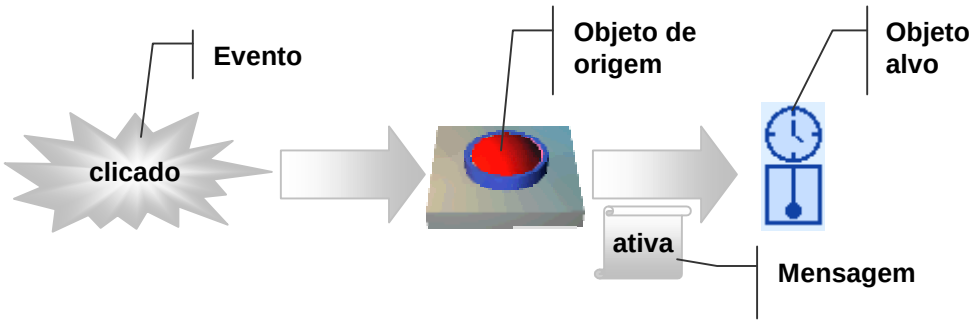


Figura 4-8: Diagrama do processo de envio de mensagem, baseado na ocorrência de um evento.

Cada objeto pode ter diversos elementos de associação e pode, eventualmente, enviar uma mensagem para si próprio; este mecanismo é utilizado para associar uma ação do objeto a um evento seu.

A interligação entre objetos é registrada dentro do espaço de trabalho conforme o *Link Schema* (Apêndice B.2). Cada interligação (*link*) fica externa aos objetos, mas faz referência ao objeto de origem e ao objeto alvo.

Apresentaremos a seguir o modelo de representação das informações estáticas (classe, interface e metadados) associadas aos objetos.

4.2.2. Representação de Classe, Interface e Metadados (CIM)

Enquanto a instância do objeto pode ser descrita genericamente, independente de plataforma e implementação, o mesmo não pode ser afirmado para a classe. Alguns aspectos da classe, como a estrutura dos atributos, herança e interfaces podem ser representadas genericamente por esquemas, no entanto, a implementação dos métodos deve ser feita com código em linguagem de programação específica.

A escolha de uma linguagem de programação implica também a escolha de:

- Um formato de estruturação do código, em termos de módulos, classes, pacotes, bibliotecas, etc.
- Uma implementação específica. Salvo as linguagens que detêm um padrão universal.
- Uma plataforma específica. Salvo para as linguagens de código móvel (MCL), ou linguagens capazes de ser recompiladas em outras plataformas sem adaptação do código fonte.
- Um conjunto de ferramentas para codificação, compilação, depuração, administração de projetos, geração automática de código, etc.

Todos estes aspectos nos levam a crer que não vale a pena definir uma linguagem única, como proposta universal, exigindo que os programadores que quiserem trabalhar com *Anima*, abram mão de outras linguagens por eles usadas. Por este motivo, *Anima* não especifica uma linguagem de programação única para a codificação da estrutura e comportamento de seus objetos.

Como conseqüência, a tarefa de integrar uma composição *Anima*, onde o estado dos objetos e seus *links* são representados em um formato neutro, com a implementação de seu comportamento feita em uma linguagem de programação, não pode se processar através da inclusão do código que define o comportamento de cada objeto, diretamente dentro da representação serializada onde está registrado o estado do respectivo objeto, tal como acontece no OOXML [SIL98] (apresentado no capítulo 3).

No caso de OOXML, utiliza-se uma única linguagem de programação, o Lisp. Por este motivo, o sistema responsável pela leitura e processamento dos seus objetos, está habilitado a carregar os trechos de código Lisp, dispostos dentro destes objetos, inserindo-os no contexto de um sistema mais amplo, onde serão interpretados. Isso é possível, tendo-se em vista que o sistema conhece o modo específico como o Lisp organiza suas unidades de código.

Anima, porém, não se limita a uma única linguagem de programação. Por isso, a sintaxe distinta das linguagens – principalmente no que diz respeito à estratégia particular de cada uma para organização das unidades de código, tais como: pacotes, classes, métodos, etc. – tornaria a inserção de código, diretamente dentro da representação serializada de estado dos objetos, uma tarefa demasiadamente complexa.

Cada trecho de código disposto na versão serializada dos objetos precisaria ser inserido em um contexto mais amplo de um sistema, elaborado na respectiva linguagem de programação, antes de ser executado. Em Java, por exemplo, as instruções de programação estão dispostas dentro de métodos. Se este método estivesse inserido dentro da representação serializada de um objeto, antes de ser executado ele precisaria ser inserido no corpo de uma classe, dado que isto é obrigatório em Java e que o método utiliza recursos deste contexto, tais como: atributos, outros métodos, etc..

Por este motivo, em *Anima*, toda a estrutura de código de programação responsável pelo comportamento dos objetos é mantida independente, no formato nativo definido pela linguagem utilizada. Para interligar o código com a representação da composição, são utilizados mecanismos de conversão, tratados mais adiante.

Entre as metas de *Anima* está a integração de objetos produzidos em linguagens de programação diferentes. Este é um problema complexo, abordado parcialmente neste trabalho. Entre os problemas encontrados, podemos citar:

- A dificuldade de se encontrar um ambiente capaz de executar objetos produzidos em linguagens de programação diferentes que interajam entre si.
- Cada linguagem ou sistema que dispõe de interface gráfica, geralmente, atua de forma exclusiva no seu espaço visual, tornando difícil a tarefa de sobrepor ou fazer interagir dois objetos de diferentes linguagens ou sistemas.
- Representação de dados e protocolos diferentes de comunicação entre objetos.

Para contornar ou solucionar parcialmente estes problemas, foram definidas as seguintes estratégias:

- Quando se trata de integrar aplicações visuais, o navegador *Web* tem-se mostrado uma opção muito boa. De fato, conforme foi abordado, programas podem ser referenciados a partir de páginas HTML. Muitas linguagens permitem de um modo ou de outro a execução de seus programas em navegadores, seja através de uma máquina virtual embutida neles, tal como Java, ou por meio de um *plugin*. Assim, a página HTML pode ser construída de tal forma, que mais de um programa possa compartilhar o espaço da página em um navegador *Web*.

- Os espaços de trabalho estabelecem áreas retangulares limitadas onde atua um conjunto de objetos (vide Figura 4-9). Conforme a caracterização de seu tipo, pode-se associar a cada área de trabalho uma linguagem de programação diferente. É possível estabelecer um ambiente de integração definindo o espaço de trabalho de mais alto nível do tipo HTML, deste modo, quando as representações são convertidas para implementações específicas, as delimitações das áreas de trabalho subordinadas se convertem na área utilizada pela máquina virtual (VM – *Virtual Machine*) ou *plugin* para a execução. Como já foi exposto, um espaço de trabalho só pode estar subordinado a outro, se eles forem compatíveis. Se o espaço de trabalho hierarquicamente mais alto for HTML, por exemplo, ele só aceitará espaços subordinados, associados a linguagens que ofereçam suporte à execução no navegador *Web* (Java, Alice, Squeak, etc.).

- Para resolver o problema da comunicação entre objetos de diferentes linguagens, adotamos um formato aberto de transmissão de mensagem, que pode ser facilmente implementado sobre qualquer suporte de comunicação, como canais TCP/IP, CORBA, etc. Este assunto é tratado em detalhes mais adiante.

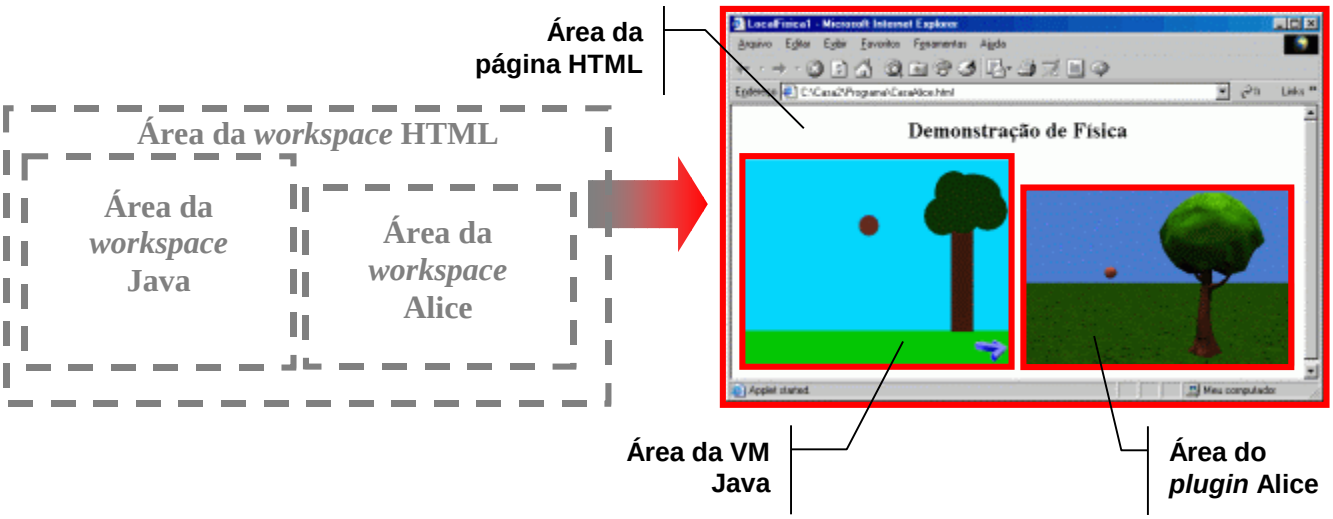


Figura 4-9: Mapeamento das áreas de trabalho em áreas utilizadas pelos programas.

Estas estratégias não são apresentadas como solução de integração entre objetos para todos os contextos, mas se mostram suficientemente eficazes para a maior parte da demanda de integração encontrada. Além disso, o projeto *Anima* foi concebido sobre uma estrutura que poderá ser facilmente estendida para dar suporte a projetos futuros, visando solucionar questões de integração não resolvidas neste trabalho.

A estratégia apresentada para utilização do navegador *Web* como ambiente de integração das aplicações nem sempre é o que desejamos. Adicionalmente, apenas algumas linguagens dão suporte a este tipo de ambiente. De qualquer modo, este caminho oferece uma solução bem interessante no contexto da educação. O projeto ESCOT, por exemplo, utiliza o navegador *Web* com sucesso, em atividades educacionais que integram diferentes componentes, contanto que todos estes sejam em Java [ROS98].

Quando combinamos integração em páginas *Web* com linguagens de código móvel, obtemos um resultado bastante portável, que pode ser distribuído ou compartilhado pela Internet.

Apesar do navegador *Web* nos parecer hoje o melhor ambiente de integração, ele não é o único, existem outras alternativas tal como a proporcionada pela linguagem Java. Algumas linguagens têm criado versões de sua implementação em Java, tais como Jacl [JOH98] – versão de Tcl – JPhyton [HUG97] – versão de Phyton – e Rhino [MOZ00] – versão de ECMAScript implementado pela organização Mozilla. Desta maneira, é possível integrá-las sobre a plataforma Java.

A mesma estratégia apresentada para integração em navegador *Web* pode ser utilizada para outras modalidades de integração, tal como Java.

A codificação de estrutura e comportamento em linguagem nativa, associada à representação da composição, é suficiente para a execução da aplicação, no entanto, do ponto de vista da montagem e da edição da composição constatamos algumas carências:

- As informações disponíveis nos arquivos de código de linguagens de programação estão formatadas principalmente para uso do compilador. Dados relativos à documentação do código, que são especialmente orientados para o programador ou usuário, na maioria dos casos, não possuem nenhum formato especial que permita sua classificação e estruturação automática. Deste modo, não há um processo formal para determinar se um comentário descreve: a funcionalidade de uma classe, o papel de um atributo ou o comportamento de um método, por exemplo. Algumas linguagens como Java são exceções a esta regra, mesmo assim, no caso específico de Java, a estrutura dos comentários possui uma configuração voltada à produção automática de documentação.
- Grande parte das linguagens não dispõe de informações suficientes para lidar com componentes de *software*. Especialmente os componentes destinados à montagem por ferramentas visuais.
- Uma ferramenta para a montagem e edição de composições, que combine mais de uma linguagem de programação, se tornaria muito complexa, dado que cada linguagem possui uma estrutura específica.

Concluímos, deste modo, que seria necessário ao modelo de *Anima* a criação de uma representação padronizada responsável pela descrição das classes e suas interfaces (não apenas do ponto de visto dos objetos, como também dos componentes) denominada CIM (Classe, Interface e Metadados).

CIM não implementa a classe propriamente dita, portanto, não dispõe de nenhum código em linguagem de programação. Ela se comporta como um conjunto de metadados de classes. As classes ficam em um recurso independente, conforme formato estabelecido pela linguagem de programação específica.

A representação CIM abrange os seguintes aspectos:

- Descrição de características da classe relevantes para o projeto de composições, tais como: identificação, herança, atributos, operações, etc.
- Descrição de características específicas de componentes, voltadas a ferramentas de edição visual.
- Declaração dos tipos de mensagens que as classes podem receber ou enviar, a fim de se estabelecer a associação entre objetos, e declaração de operações associadas a mensagens recebidas.
- Metadados, incluindo os definidos pelo padrão LOM [HOD02].

Em termos de descrição da classe, a representação CIM não se limita a identificadores e parâmetros definidos pelas linguagens de programação Ela agrega uma estrutura para descrição detalhada de cada elemento da classe, inclusive com suporte em múltiplos idiomas. Os metadados educacionais, conforme o padrão LOM, permitem a identificação de aspectos pedagógicos de cada objeto, que orientam sua seleção e uso na produção de composições. Além disso, eles estabelecem uma estrutura padronizada de documentação e classificação, de especial importância para mecanismos de busca e bibliotecas de objetos e componentes.

A estrutura da representação CIM está definida no CIM *Schema* (Apêndice B.2).

Em paralelo com a definição do esquema de representação CIM, foi organizada uma metodologia para a sua produção, manutenção e uso. Esta metodologia está dividida em algumas etapas que, apesar de não serem todas obrigatórias nem necessariamente executadas na ordem apresentada, compõem o processo adotado para a concepção da representação CIM.

A seguir, apresentamos as etapas da metodologia:

A. Importação da estrutura da classe por reflexão

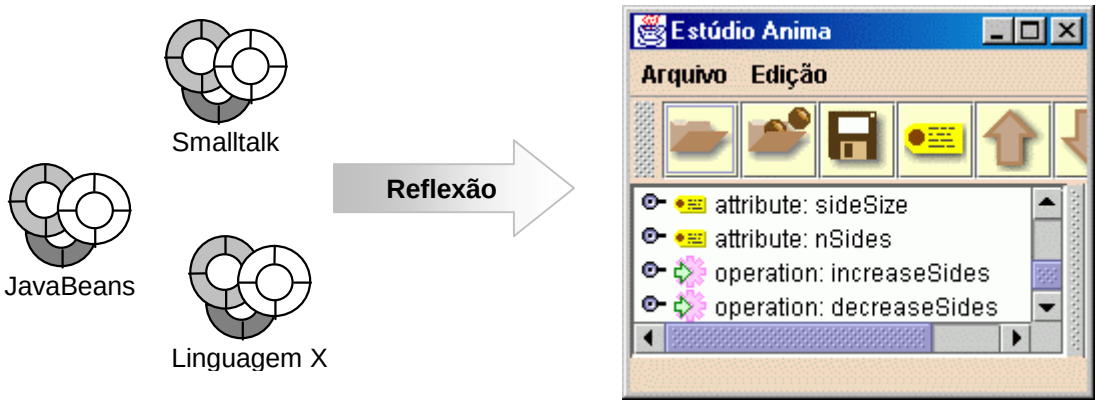


Figura 4-10: Importação da estrutura de uma classe utilizando o mecanismo de reflexão.

Os dados básicos para a construção da representação CIM podem ser extraídos automaticamente da codificação das classes (Figura 4-10). Isto é possível graças a recursos de metaprogramação e reflexão (*reflection*) disponíveis em muitas linguagens de programação.

François-René Rideau define metaprogramação como "a arte de programar programas que lêem, transformam, ou gravam outros programas". Seguindo esta lógica, ele define que sistemas ou linguagens reflexivos proporcionam a seus programas alguma funcionalidade especial de ler ou gravar a si próprios (metaprogramação reflexiva) [RID99].

Christoph Steindl atribui a Smalltalk e Lisp o pioneirismo em metaprogramação e reflexão. Smalltalk, em especial, descreve a estrutura de suas classes em metaclasses, cujas informações podem ser acessadas e modificadas [STE96]. Posteriormente, este recurso se estendeu a muitas outras linguagens, como Java (Java *Reflection*), C++ (RTTI), etc.

Através da reflexão, uma das ferramentas de *Anima* acessa informações da classe – como: propriedades, métodos, eventos a que responde, etc. – diretamente no arquivo onde ela está codificada. Estas informações são a base, a partir da qual as descrições são posteriormente refinadas.

B. Modelagem e edição da representação

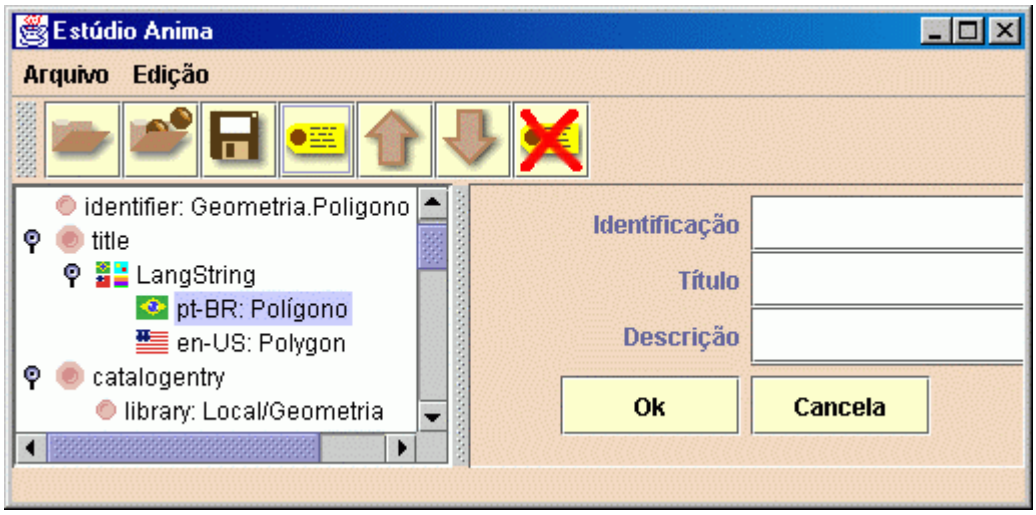


Figura 4-11: Modelagem e edição de propriedades da classe e interface.

A próxima etapa consiste no refinamento da representação CIM, através de uma ferramenta de modelagem e edição (Figura 4-2, indicador ⑧). Apesar da metodologia sugerir que o processo se inicia na importação de características da classe, através da reflexão, essa etapa não é obrigatória. A ferramenta de edição pode ser utilizada para a criação da

representação a partir da estaca zero. Isto pode ser especialmente conveniente para linguagens que não possuem recursos de reflexão, ou cujo processo não esteja implementado na ferramenta.

Entre as tarefas desta ferramenta, destacamos:

- Refinamento dos dados importados por reflexão, tal como a descrição detalhada, e eventualmente em mais de uma língua, da classe, atributos, métodos, etc.
- Estruturação das interfaces, definindo propriedades acessíveis ao usuário durante o processo de edição visual, tipos de mensagens associadas a eventos, ou tipos de mensagens que podem ser recebidas por um objeto desta classe.
- Descrição de metadados, especialmente os educacionais conforme o padrão LOM.

C. Armazenamento da representação

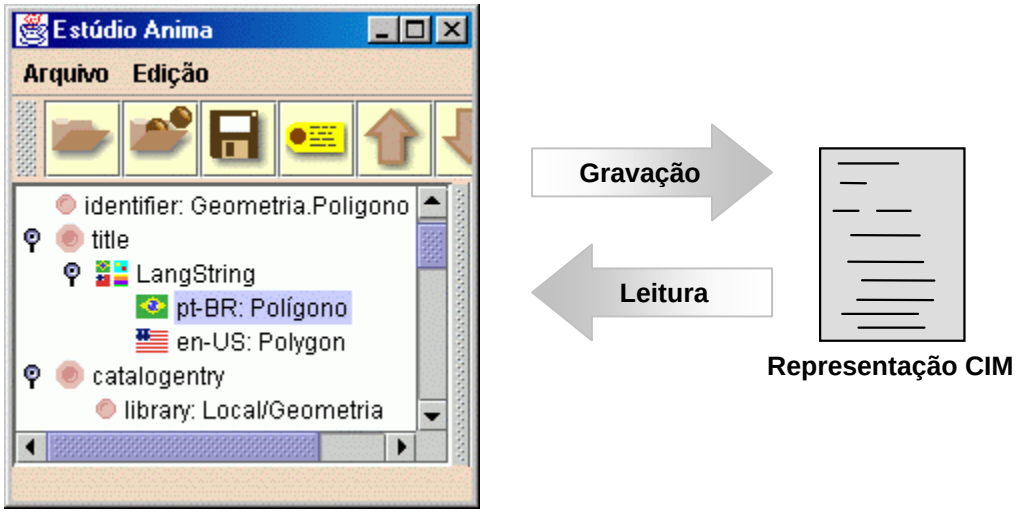


Figura 4-12: Gravação e leitura da representação CIM em formato aberto.

A ferramenta de modelagem e edição deve gravar seu resultado em um arquivo de representação CIM, cujo formato é padronizado (segundo CIM Schema), aberto e independente de plataforma ou implementação.

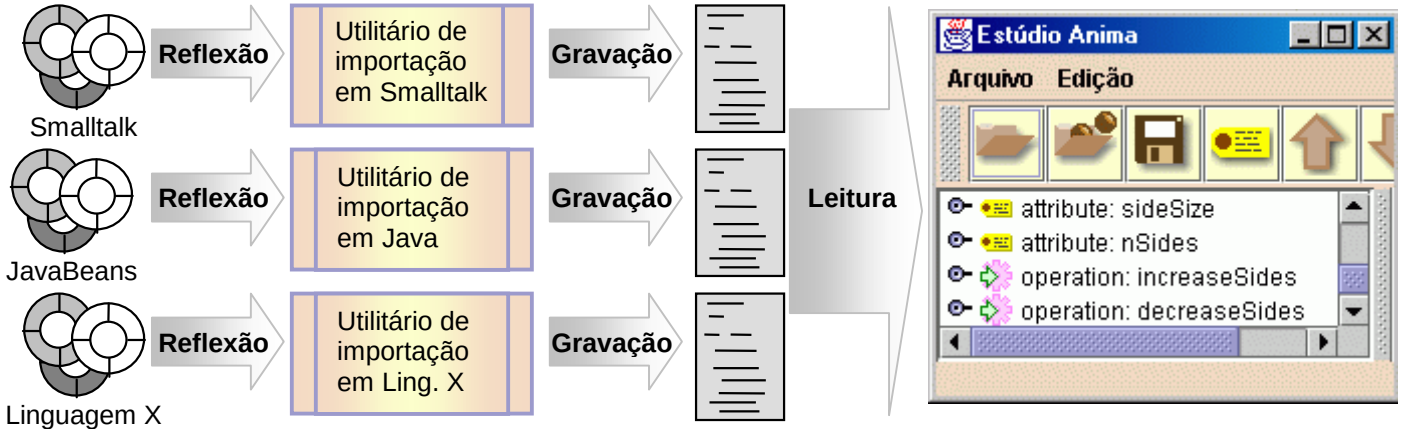


Figura 4-13: Processo de importação realizado de forma independente da edição.

Este arquivo cumpre diversas funções:

- É utilizado como meio de armazenamento, permitindo que a ferramenta de modelagem e edição realize futuras modificações na representação.
- É utilizado por outras ferramentas que necessitam da descrição CIM.
- Permite a separação entre a importação por reflexão e a edição (vide Figura 4-13). Isto significa que para cada linguagem de programação pode haver um utilitário, escrito na mesma linguagem (Figura 4-2, indicador ⑦), que importe as estruturas de suas classes. Estes utilitários gravam o arquivo CIM padrão, que pode ser lido por uma única ferramenta de modelagem e edição.

D. Composição baseada em CIM

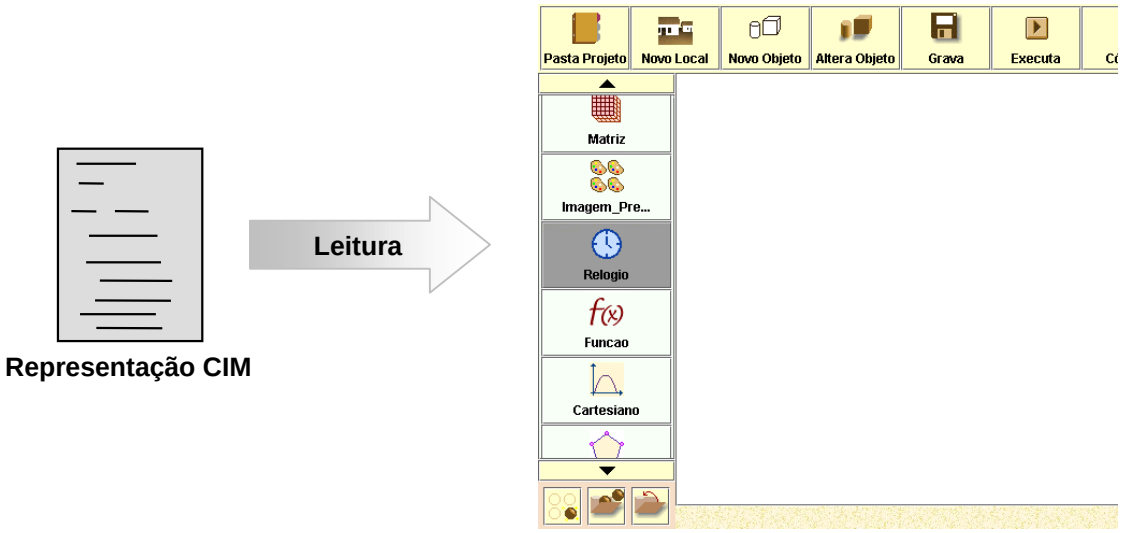


Figura 4-14: Leitura de representação CIM por uma ferramenta de montagem e edição de composição.

Qualquer ambiente destinado à montagem e edição de composições[2] (Figura 4-2, indicador ⑥) poderá facilmente realizar a leitura de uma representação CIM, uma vez que é um formato aberto e independente de plataforma e implementação.

A representação CIM possui um conjunto de informações completo, para que o ambiente possa apresentar e manipular todos os componentes disponíveis para a produção (vide Figura 4-14), sem a necessidade do acesso à implementação dos mesmos. Para apresentá-los, a ferramenta dispõe de informações descritivas e referências a arquivos de imagem, que contêm ícones dos componentes.

No momento em que o usuário seleciona um determinado componente, o ambiente decide se está habilitado a instanciar um objeto na linguagem em que a respectiva classe foi escrita. Isto depende da compatibilidade entre a plataforma da ferramenta e a da classe. Ambas podem estar na mesma plataforma, em plataformas compatíveis ou em plataformas que possuam interfaces que lhes permita trabalhar em conjunto.

Como pode ser observado na Figura 4-15, se for possível gerar uma instância do objeto, o usuário poderá, durante o processo de produção, manipular diretamente o componente, configurá-lo e experimentar seu comportamento.

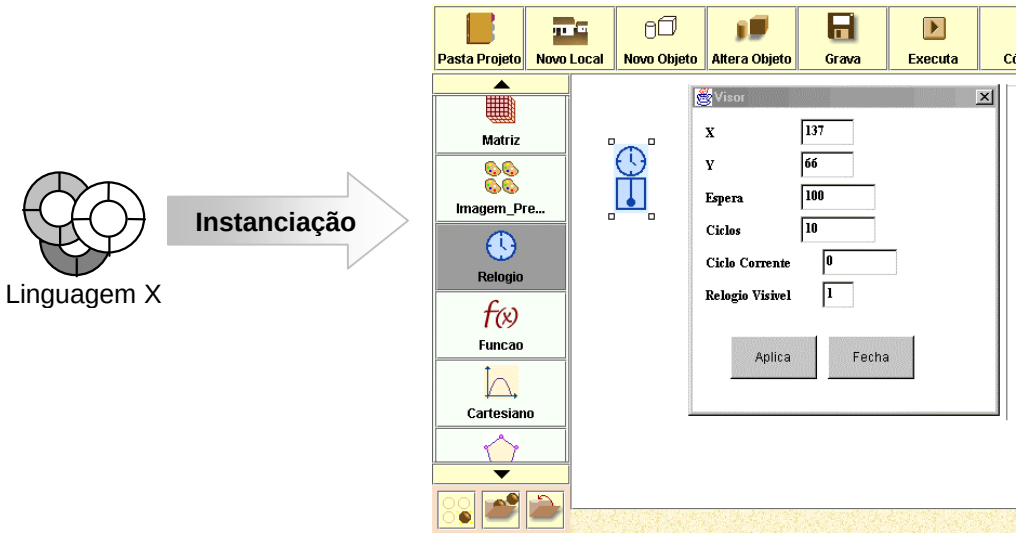


Figura 4-15: Instanciação de componente de classe compatível com ambiente de produção.

Caso contrário, o ambiente ainda tem acesso, a partir da representação CIM, a um conjunto suficiente de informações sobre a interface e apresentação do objeto, que possibilitam sua inserção e integração na composição, mesmo que ele não possa ser experimentado interativamente. Através de uma imagem (definida na representação CIM) que

[2] Estamos utilizando aqui, como em todo o texto, o termo "composição" para representar um subconjunto de aplicações formadas de objetos, conforme estrutura definida por *Anima*. As composições são discutidas no tópico 4.2.1.

representa o componente, o usuário poderá definir sua localização e espaço físico. CIM ainda dispõe de informações referentes a propriedades que podem ser configuradas no componente e seu tipo, eventos a que ele responde, mensagens que pode enviar e receber, etc.

O processo de montagem e edição da composição é tipicamente realizado através de uma ferramenta visual, que trata todos os detalhes de funcionamento e representação de *Anima*. O usuário não precisa possuir qualquer conhecimento de *Anima* para operar com a ferramenta.

E. Armazenamento da composição

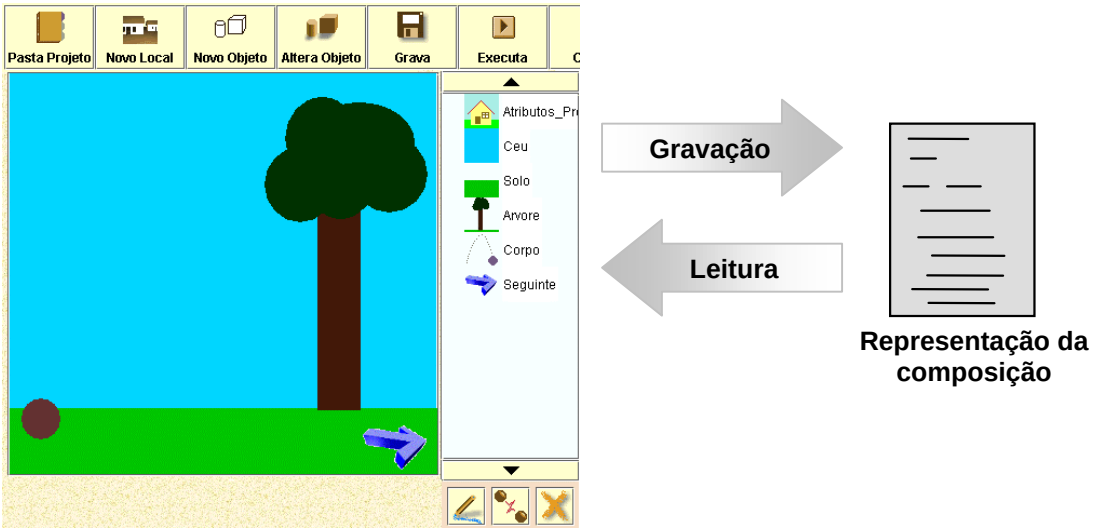


Figura 4-16: Gravação e leitura da composição em arquivo.

O produto final da composição, com seus espaços de trabalho, objetos e interligações é então armazenado em um arquivo padrão, aberto, independente de plataforma e implementação (Figura 4-16), conforme esquemas definidos no tópico 4.2.1.

Este arquivo cumpre as seguintes funções:

- É utilizado como meio de armazenamento, permitindo que o ambiente realize futuras modificações na composição.
- É utilizado para a execução da aplicação resultante, conforme será apresentado adiante.

Para compreender o modo como uma representação CIM está organizada, é necessário esclarecer alguns aspectos:

- Cada recurso CIM descreve uma única classe.

- Alguns elementos da descrição correspondem a um mapeamento de características da classe, tais como: superclasse, linguagem de programação, estrutura de componente, etc.
- Alguns elementos de descrição cumprem a função de documentação: títulos, descrições, metadados educacionais, etc.
- Alguns elementos de descrição são especialmente projetados para uso em ambientes de produção visual: ícone, fotografia do objeto (imagem que substitui objeto, caso a ferramenta não possa instanciá-lo), visões, etc.
- Cada operação da classe possui uma ou mais mensagens associadas, que ao serem recebidas pelo objeto disparam a operação. São definidos os tipos de mensagens suportados, de modo que se possa verificar a compatibilidade entre a mensagem enviada por um objeto e a recebida por outro.
- Além de mapear os atributos e operações da classe, CIM relaciona as propriedades. Um dos princípios da orientação a objetos prevê que um objeto não deve ter acesso direto aos atributos de outro. Deste modo, as linguagens têm criado mecanismos de acesso indireto, cada uma a seu modo (seja através de métodos, seja através da exportação de atributos com regras de restrição de acesso, etc.), que denominamos propriedades. Existem propriedades que simulam atributos, mas não estão associadas a nenhum deles.
- O modelo de *Anima* prevê que o mesmo objeto possa ser tratado por diversos pontos de vista, seja em grau diferente de complexidade ou associados a diversos domínios. Estas diferentes "visões" do objeto são cadastradas como o elemento View. Cada visão (*view*) define quais os atributos, métodos e propriedades serão apresentados ao usuário.
- A representação CIM relaciona o conjunto de eventos que os objetos de uma classe capturam, determinando o tipo de mensagem que pode ser produzida na ocorrência do mesmo. Esta informação é utilizada na interligação de objetos. O ambiente de produção deve estabelecer uma compatibilidade entre a mensagem produzida e a recebida.

O próximo passo é definir o processo que transforma este conjunto de informações genéricas em uma implementação específica, a ser discutida no tópico a seguir.

4.2.3. Mecanismos de conversão

Os mecanismos de conversão são a ponte entre o modelo representado genericamente e sua versão equivalente, implementada em uma linguagem (ou conjunto de linguagens), pronta para ser executada no computador.

Para realizar esta conversão, o projeto Anima faz uso de ferramentas, também independentes de plataforma, mantendo a coerência da estrutura como um todo.

Como ilustrado na Figura 4-17, o plano de transformação é associado aos recursos de composição. Um mecanismo de transformação (Figura 4-2, indicador ⑨) realiza sua conversão para um novo produto resultante (Figura 4-2, indicador ⑩), observando duas regras de conduta:

- Cada linguagem destino para conversão, possui um plano de transformação diferente, de modo que a mesma representação de entrada pode gerar diferentes resultados na saída.
- A conversão é feita, a princípio, para a linguagem especificada no espaço de trabalho. Eventualmente, o usuário pode solicitar a conversão em outra linguagem – o mecanismo de transformação verifica se é possível, conforme compatibilidade entre a linguagem definida no espaço de trabalho e a solicitada, operando a conversão quando possível.

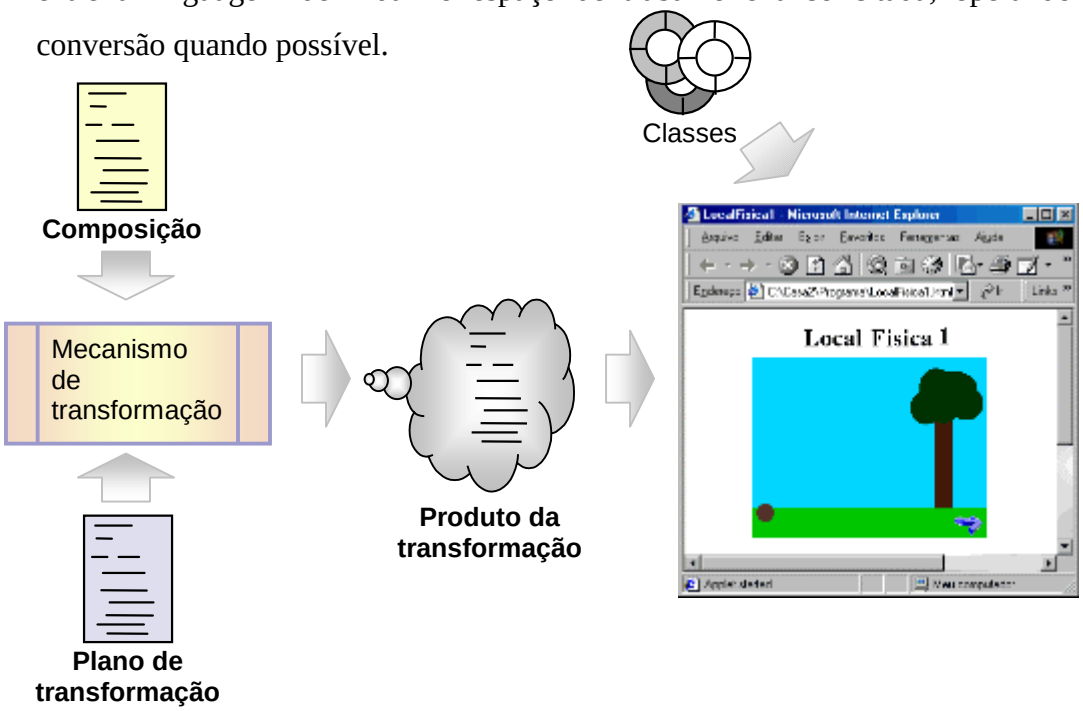


Figura 4-17: Diagrama de transformação das representações genéricas em uma aplicação implementada em linguagem específica.

Para converter uma composição genérica na respectiva aplicação em linguagem e implementação específica, distinguimos dois caminhos:

- Construir um programa de transformação que detém toda a lógica do processo. Este programa só necessita do recurso de composição e gerar o produto final em uma linguagem específica. Para cada linguagem ou solução diferente, torna-se necessário um novo programa. Algumas propostas apresentadas, tais como Coins e JAXB, utilizam esta solução.
- Construir um programa genérico de transformação que, baseado em um arquivo que detém o "plano de transformação", realiza a conversão. Para cada linguagem ou solução diferente utiliza-se um novo "plano de transformação". No entanto, o programa é sempre o mesmo. O plano é escrito em formato aberto e flexível o suficiente para se adaptar a qualquer necessidade.

Esta segunda solução é a proposta de *Anima*.

Atualmente existem linguagens capazes de representar planos de transformação independentes de plataforma e implementação. Este é o caso da DSSL – *Document Style Semantics and Specification Language* [ISO96] – linguagem baseada em SGML – *Standard Generalized Markup Language* [ISO86] – e da XSL – *Extensible Stylesheet Language* [ADL01] – linguagem baseada em XML.

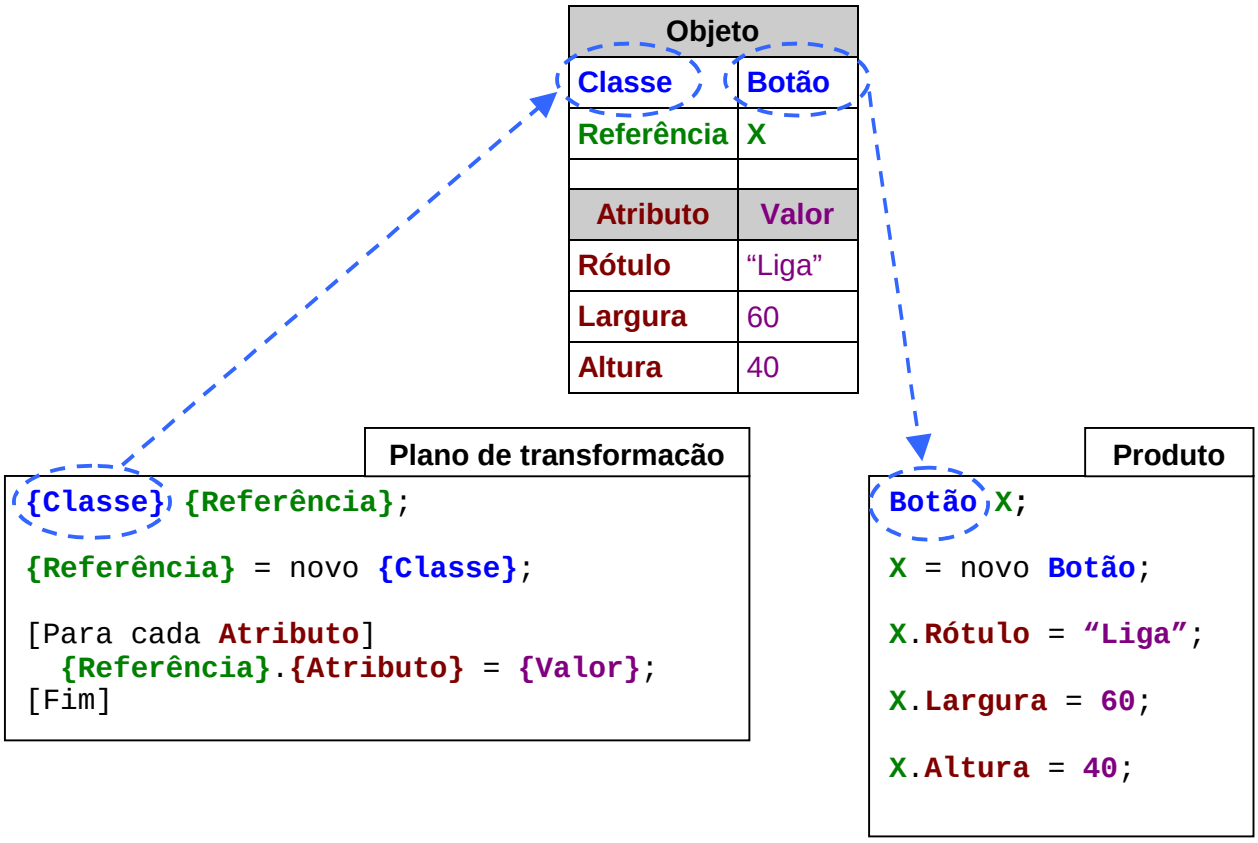


Figura 4-18: Aplicação de plano de transformação na representação de um objeto.

A Figura 4-18 ilustra um dos princípios básicos do plano de transformação. Sua função é estabelecer como cada elemento da composição será mapeado para o produto final.

Este exemplo apresenta um plano simples, que faz uma substituição de campos. No entanto, é possível se estabelecer planos mais sofisticados, capazes de relacionar mais de um recurso ou de definir condições e estruturas de seleção.

Dependendo do formato do produto, será necessário recuperar dados não apenas do recurso de composição, como também do CIM. O tipo dos atributos pode ser importante para a decisão de como eles serão montados no arquivo de resultado. Em Java, por exemplo, um literal String é representado dentro de aspas duplas e um literal numérico não possui aspas.

4.2.4. Protocolo padronizado de comunicação

Realizar a comunicação entre os objetos foi o desafio final deste projeto.

O problema central consiste em promover a comunicação entre objetos que possuem formatos diferentes de comunicação de dados em tempo de execução, seja porque estejam

codificados em linguagens diferentes, ou porque, ainda que estejam na mesma linguagem, trabalhem com formatos incompatíveis.

A Figura 4-19 apresenta um diagrama que sintetiza a estrutura de comunicação utilizada por *Anima*.

Inicialmente é necessário entender que *Anima* estabelece um formato padrão de mensagens trocadas entre os objetos que, por ser baseado sempre em uma cadeia de caracteres Unicode, pode ser tratado igualmente por qualquer linguagem de programação que aceite Unicode. Números e outros tipos de dados que utilizam uma representação binária específica e diferenciada em cada linguagem ou implementação transformam-se também em cadeias de caracteres, ao serem convertidos para o formato *Anima*.

Além de estabelecer que toda mensagem se baseia em texto, *Anima* define um formato padrão para a representação textual da mesma, conforme o *Message Schema* (Apêndice B.3).

Este esquema define um conjunto de formatos pré-definidos para as mensagens mais comuns – organizados na forma de categorias – mas permite também um refinamento deste formato, através de esquemas elaborados pelo desenvolvedor, que estabelecem:

- quantidade de parâmetros;
- tipo dos parâmetros;
- nome de cada parâmetro ou ordem em que eles aparecem.

O esquema de refinamento deve seguir o plano de um meta-esquema, que estabelece seus elementos. Este meta-esquema é apresentado no Apêndice B.3 com o título de: *Schema for Type Message Schema*.

Ao contrário do formato padronizado de *Anima*, cada linguagem possui seu formato específico de mensagem. Portanto, faz-se necessária a criação de uma interface que realize a conversão entre o formato utilizado pela linguagem e o formato *Anima*. O diagrama da Figura 4-19 destaca uma interface entre cada objeto e o ambiente intermediado por *Anima*. O papel da interface é receber mensagens em formato interno e despachá-la externamente em formato Anima e vice-versa (receber mensagem externa em formato *Anima* e enviá-la ao objeto em formato interno).

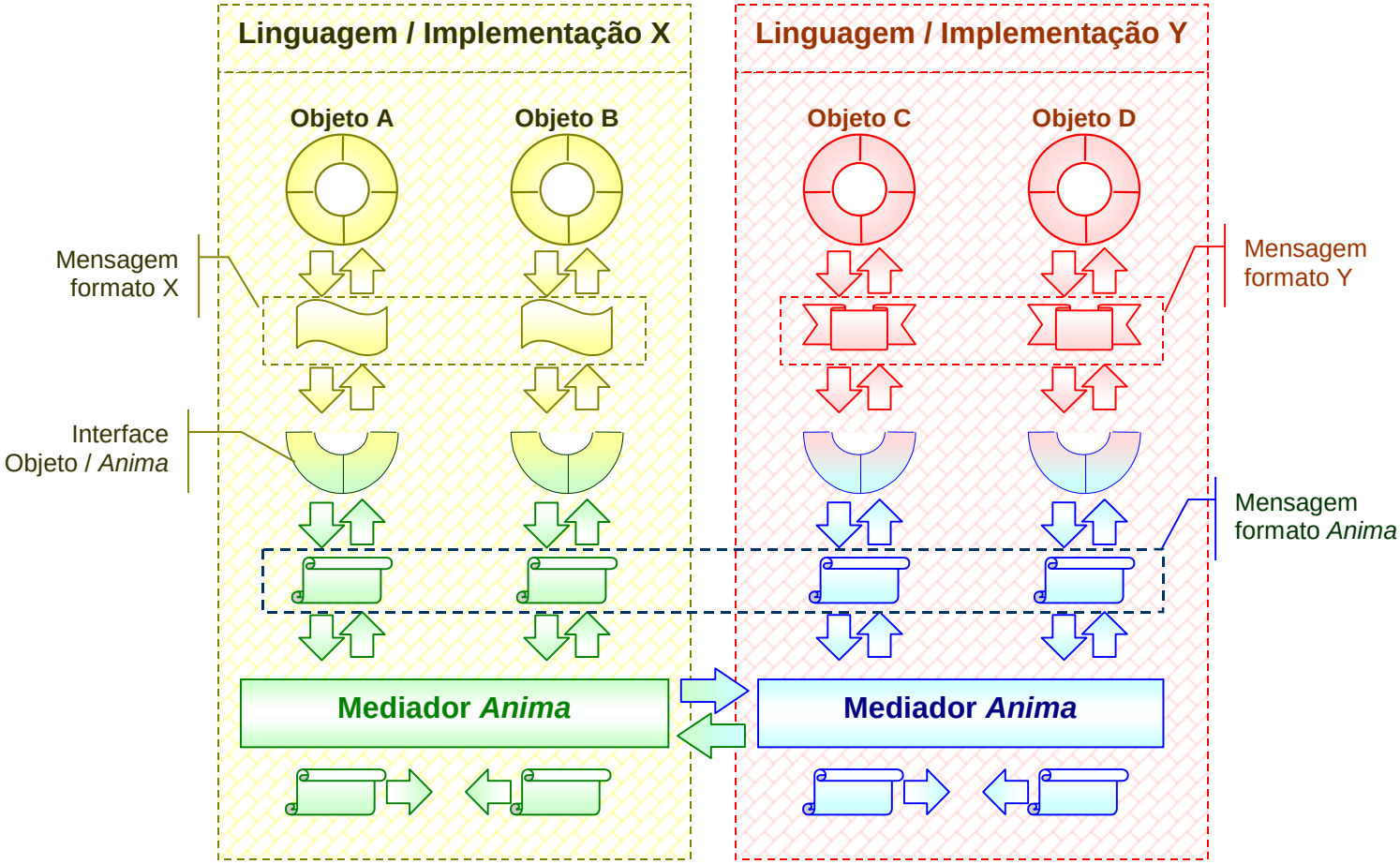


Figura 4-19: Diagrama de comunicação entre objetos através de *Anima*.

Além de realizar a conversão de formatos de mensagem, a interface está habilitada a se comunicar com o Mediador *Anima*. O Mediador possui as seguintes funções:

- registrar cada objeto do espaço de trabalho a partir da sua identificação *Anima* (utilizada na serialização dos objetos);
- conduzir a mensagem da sua origem até o seu destino;
- comunicar-se com outros Mediadores a fim de permitir a troca de mensagens entre espaços de trabalho diferentes.

Como ilustra a Figura 4-20, a interface do objeto com o ambiente Anima pode ser construída de três formas distintas:

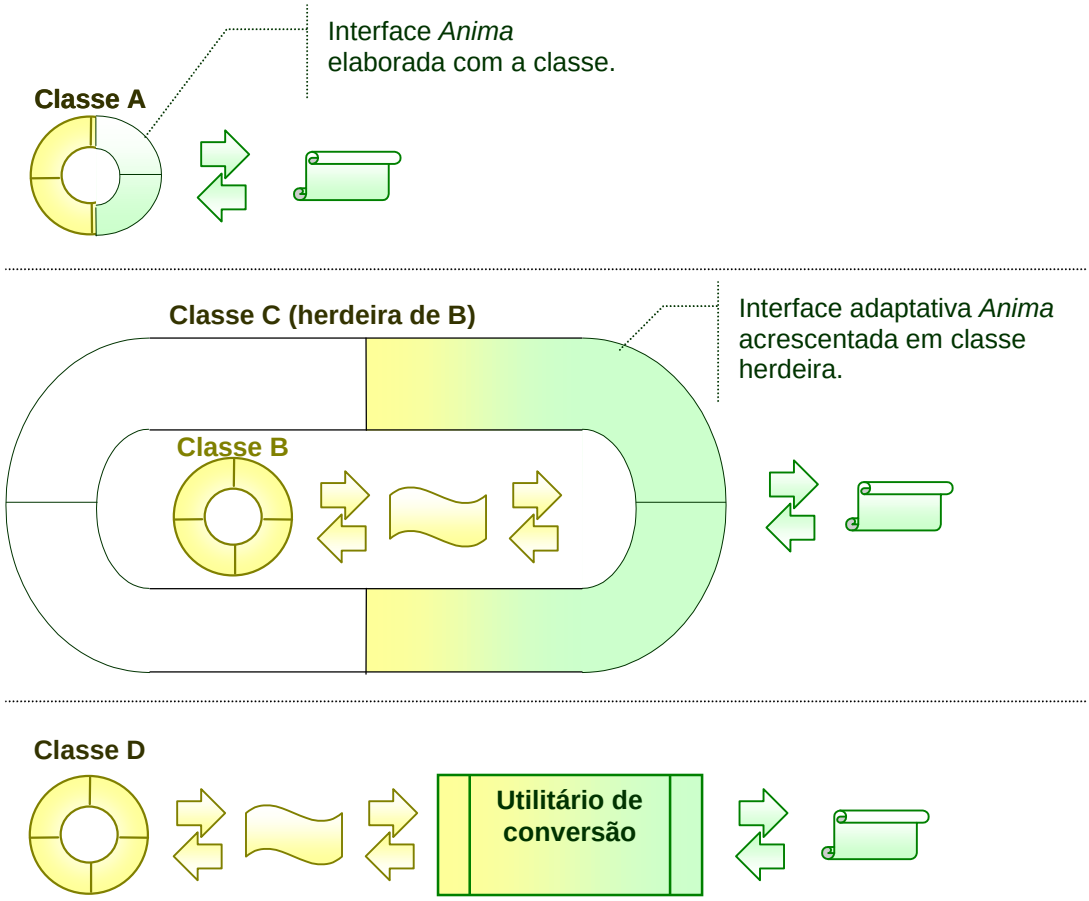


Figura 4-20: Modalidades de construção de interface entre objeto e *Anima*.

Na primeira solução, o próprio programador ao construir a classe elabora uma interface conforme o padrão *Anima*. Neste caso, os objetos da classe já estão habilitados a se comunicar nativamente com o ambiente *Anima*. Para a construção desta interface, o programador irá dispor do suporte de bibliotecas *Anima* na linguagem nativa, com facilidades de conversão de mensagem, conexão com o ambiente, etc.

A segunda solução considera uma classe pré-existente que deverá ser adaptada para o ambiente *Anima*. Sem alterar a classe original (B), cria-se uma classe herdeira (C) que acrescenta uma interface com o ambiente *Anima*. A vantagem desta segunda solução é que ela pode ser produzida automaticamente por uma ferramenta que, acessando a estrutura interna da classe por reflexão e tendo acesso à representação CIM da mesma, é capaz de produzir automaticamente código em linguagem nativa que faça as respectivas conversões.

Na última solução não é realizada qualquer intervenção durante o processo de codificação da classe. Durante a execução da composição, toda a mensagem enviada ou recebida por objetos de uma determinada classe (D), é capturada automaticamente por um utilitário de conversão, que converte a mensagem e todos os parâmetros que a acompanham.

A conversão durante a execução acrescenta um *overhead* na mesma, pois o utilitário precisa utilizar mecanismos de reflexão para determinar o modo como deve se comunicar com a classe e possuir um algoritmo genérico de conversão de mensagens, baseado em informações obtidas na representação CIM em tempo de execução.

Os Mediadores são construídos na mesma linguagem e implementação dos objetos do espaço de trabalho. Se por um lado, as mensagens trocadas entre Mediador e objeto possuem um formato padronizado, o canal utilizado para a transferência desta mensagem é aquele disponível na própria linguagem. Quando há uma interface independente entre o objeto e o Mediador, a troca de mensagens entre interface e objeto utiliza formato e canal disponíveis na linguagem, já a troca de mensagens entre interface e Mediador utiliza formato *Anima* e canal disponível na linguagem.

Os Mediadores podem manter comunicação entre si. Quatro situações podem acontecer:

- Mediadores construídos na mesma linguagem e que estão sendo executados no mesmo espaço de endereçamento, ou pela mesma máquina virtual, ou pelo mesmo módulo *runtime* ou *plugin*.
- Mediadores construídos na mesma linguagem e que estão sendo executados em espaços de endereçamento diferentes, ou por duas máquinas virtuais distintas, ou por módulos *runtime* ou *plugin* distintos.
- Mediadores construídos em linguagens de programação diferentes, ou implementações incompatíveis da mesma linguagem.
- Mediadores que estão sendo executados em computadores diferentes, interligados através de uma rede.

No primeiro caso é possível utilizar, em algumas linguagens, o próprio mecanismo de envio de mensagens utilizado internamente no programa. Existem, porém, situações em que isto não é possível.

Tanto para o primeiro quanto para o segundo caso, muitas linguagens de programação oferecem mecanismos específicos que dão suporte à troca de mensagens. Java, por exemplo, dispõe da tecnologia InfoBus, que “é uma especificação pública de uma tecnologia para intercâmbio dinâmico de dados, que permite que desenvolvedores equiparem seus componentes JavaBean, para se comunicarem com outros componentes JavaBean” [COL99].

O InfoBus define mecanismos para que os componentes troquem informações de forma estruturada. Ele é capaz de propiciar a comunicação de aplicações Java nos dois casos (primeiro e segundo).

A comunicação entre objetos de linguagens e implementações diferentes, ou a comunicação entre objetos em máquinas diferentes ligadas por uma rede são questões mais complexas, que envolvem diversos aspectos e que têm sido tratadas há algum tempo, especialmente pela OMG – *Object Management Group* [VIN97].

Comunicação entre objetos distribuídos em ambientes heterogêneos

CORBA

A OMG elaborou uma especificação denominada CORBA – *Common Object Request Broker Architecture* [VIN97] – apta à comunicação entre objetos distribuídos em ambientes heterogêneos. Os objetos podem estar escritos em linguagens de programação e implementação diferentes e podem estar dispostos em computadores diferentes.

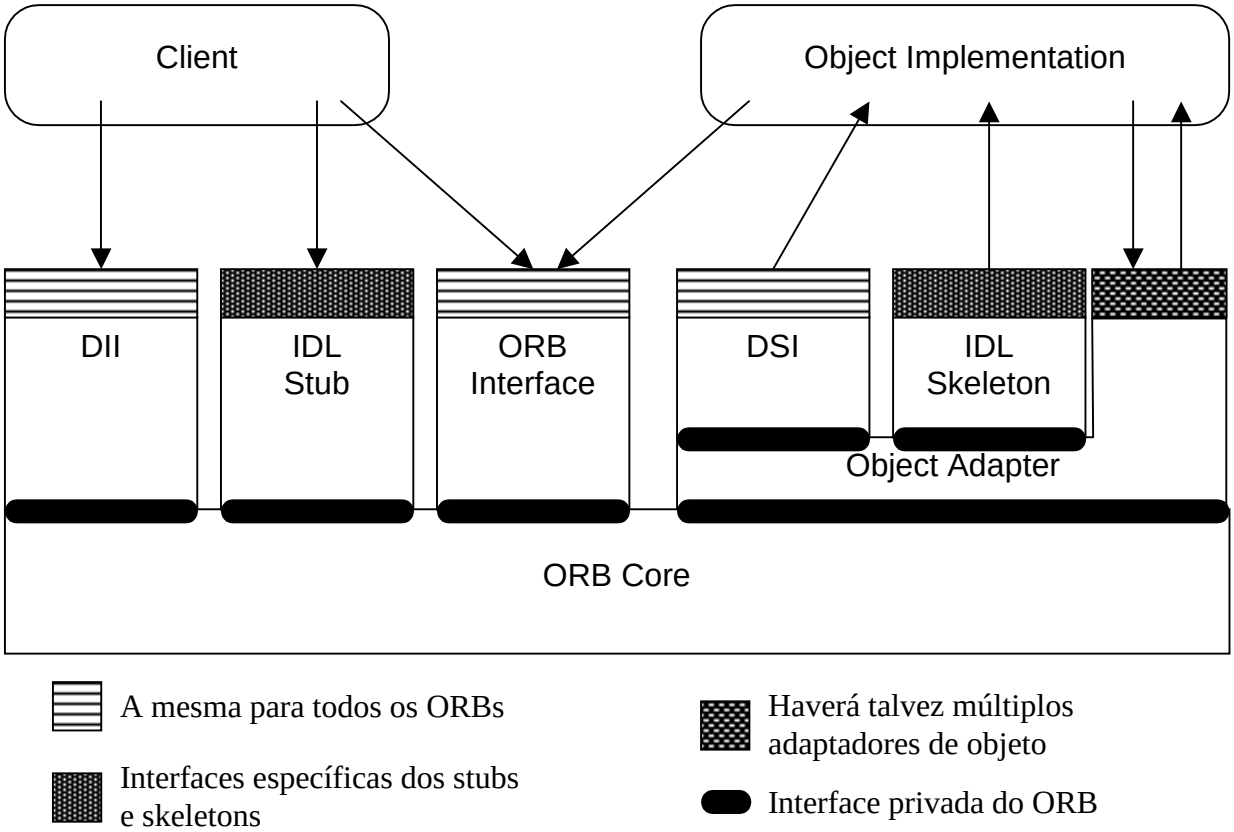


Figura 4-21: *Common Object Request Broker Architecture* [VIN97]

A Figura 4-21 ilustra o processo geral de funcionamento da arquitetura. Ela se baseia no princípio de que cada objeto (Object Implementation) disponibiliza seus serviços através

de interfaces. Estes serviços são requisitados por um cliente (Client) e toda a mediação cliente-objeto é realizada pelo ORB – *Object Request Broker*.

Interfaces disponibilizadas pelos objetos são descritas através de uma linguagem padrão denominada IDL – *Interface Definition Language*.

A comunicação cliente-objeto, mediada pelo ORB, pode se processar através de invocações estáticas ou dinâmicas.

Na modalidade estática a descrição das interfaces, escrita em IDL, é traduzida através de um programa específico para o IDL Stub e o IDL Skeleton. O Stub é codificado na mesma linguagem do Cliente, enquanto o Skeleton é codificado na linguagem do Objeto.

Toda a relação do Cliente com o Objeto é feita através de invocações locais a procedimentos implementados no Stub, que, por sua vez, se comunica diretamente com o ORB e através dele procede às ações no Objeto. Desta maneira, o cliente tem a impressão de estar interagindo com o objeto localmente; é função do Stub mapear esta interação para um universo distribuído.

O Skeleton realiza o mesmo papel do lado do Objeto, ele despacha as requisições do Objeto que, por sua vez, tem a impressão de estar interagindo com um Cliente local.

Ao contrário da modalidade estática, a dinâmica não exige a geração de código de interface específico, para cada cliente ou objeto.

O Cliente utiliza uma interface padrão denominada DII – *Dynamic Invocation Interface* – para interagir diretamente com o ORB. Através da DII o cliente pode solicitar serviços de qualquer objeto, sem ter qualquer informação de sua interface codificada em tempo de compilação. A DII acessa um repositório de interfaces IDL, de onde extrai as informações necessárias relativas à interface com o objeto. Isto implica em um *overhead* adicional na interação cliente-objeto.

Do lado do Objeto está a DSI – *Dynamic Skeleton Interface* – que desempenha papel equivalente à DII, para despachar requisições do objeto.

SOAP

Recentemente, o W3C vem liderando a padronização do SOAP – *Simple Object Access Protocol* [WIL01] – cujo objetivo também é a comunicação entre aplicações em ambientes distribuídos e heterogêneos.

O princípio básico em que se fundamenta o SOAP é a transferência de mensagens codificadas em pacotes XML, considerando a mesma situação abordada no CORBA, onde um Cliente deseja utilizar serviços oferecidos por um Objeto.

Inicialmente, o Cliente formata a requisição que será feita ao Objeto em um pacote XML, conforme esquema definido pelo SOAP. Através de um protocolo de comunicação como o HTTP, o pacote é enviado para o servidor onde está o Objeto. Uma rotina de escuta, que está sendo executada neste servidor, reconhece a solicitação SOAP e a decodifica em uma chamada de método em formato específico do objeto.

O Objeto retorna o resultado para a rotina de escuta, que a codifica em outro pacote XML/SOAP e o envia de volta para o Cliente. Este, por sua vez, decodifica o pacote e extrai a resposta desejada.

Todos os serviços oferecidos pelos objetos de um servidor ficam dispostos em um arquivo XML separado, denominado SDL (*Service Descriptor Language*).

Na comunicação entre Mediadores *Anima* escritos em linguagens ou implementações diferentes, ou entre Mediadores que estão sendo executados em máquinas diferentes, interligadas através de uma rede, faz-se necessário o uso de uma destas tecnologias para a comunicação entre objetos distribuídos em ambientes heterogêneos.

Deste modo, conforme apresentamos na Figura 4-22, os Mediadores se comunicam entre si, utilizando CORBA, SOAP, *sockets* TCP/IP ou qualquer outra tecnologia equivalente. Em um protótipo experimental que está sendo executado em um navegador *Web*, utilizamos um *socket* TCP/IP para realizar a comunicação entre dois Mediadores, escritos em linguagens de programação diferentes.

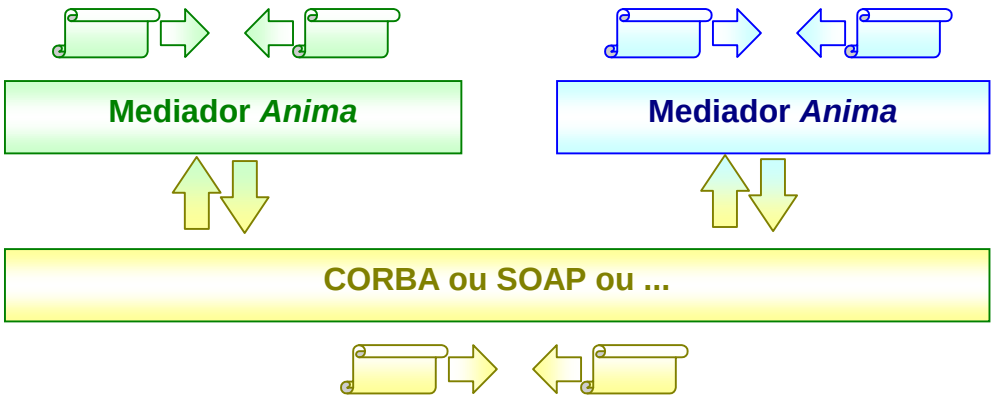


Figura 4-22: Comunicação entre Mediadores *Anima* através de CORBA, SOAP, etc.

Se um objeto de um espaço de trabalho deseja enviar uma mensagem para um objeto de outro espaço de trabalho, ele utiliza a identificação do espaço destino como prefixo da referência do objeto. Deste modo, o Mediador procura um outro Mediador associado ao espaço de trabalho especificado, se encontrar, envia para ele a mensagem. O Mediador destino recebe a mensagem e a despacha para o objeto.

Quando o Mediador utiliza uma tecnologia como CORBA, ele não mapeia a interface de todos os seus objetos em IDL. As interfaces que ficam disponíveis são a dos Mediadores, que se comportam como objetos CORBA. Quando o Mediador A despacha uma mensagem para o Mediador B, o primeiro se comporta como um Cliente CORBA e o segundo como um Objeto CORBA. O Cliente, neste caso, sempre solicitará o mesmo serviço, que consiste em despachar uma mensagem XML para o Objeto CORBA destino, que é o Mediador B.

A relação com o SOAP se faz através do empacotamento das mensagens *Anima*, que já estão em XML, dentro de pacotes SOAP.

Esta apresentação do protocolo de mensagens é o último componente do modelo *Anima*. No próximo tópico apresentamos o mapeamento de cada aspecto deste modelo para a linguagem XML e derivadas.

4.3. Representação baseada em marcação estruturada

O modelo abstrato apresentado necessita de um meio de representação adequado para seu processamento por computadores. Se, por um lado, a escolha de um formato de representação especializa o modelo, por outro, buscamos um padrão que, por ser aberto, extensível e independente de plataforma ou implementação, seja o ponto de equilíbrio entre a representação abstrata e uma implementação especializada.

Este foi o motivo da adoção de XML (Apêndice A.1). Ela é ao mesmo tempo uma linguagem que utilizaremos como suporte à representação de informações, como as de um espaço de trabalho, por exemplo, e uma metalinguagem capaz de definir novas linguagens particulares, como a que utilizaremos para mapear os esquemas definidos.

4.3.1. Representação da Composição em XML

Conforme pode ser observado na Figura 4-23, a estrutura hierárquica de XML permite representar as informações contidas em esquemas e sub-esquemas definidos no modelo *Anima* para a representação da Composição.

Classe Botão		
Id: X	Atributos	`<object id="X" class="Botão">`

Liga	Rótulo: Liga
	Largura: 60
	Altura: 40
		`<attribute name="Rótulo">Liga</attribute>`
		`<attribute name="Largura">60</attribute>`
		`<attribute name="Altura">40</attribute>`
		`</object>`

Figura 4-23: Estado de um objeto representado em XML.

Foi utilizada uma DTD – *Document Type Definition* [BRA00] – para mapear o *Workspace Schema* (Apêndice B.1) e respectivos sub-esquemas. A DTD e um exemplo de arquivo XML que a segue são apresentados no Apêndice C.1.

Não existe um conjunto de regras formais que determinem como mapear a estrutura do esquema apresentado para XML. A estrutura XML permite a escolha de soluções igualmente válidas para diversas questões de mapeamento. Robin Cover mantém uma seção intitulada "*Using Elements and Attributes*"[3], que sintetiza importantes aspectos desta discussão.

A DTD para a representação de composições define como elemento mais abrangente, ao qual todos os demais estão subordinados, o "espaço de trabalho" através do elemento `<workspace>`.

Dentro do espaço de trabalho estão dispostos os objetos a ele pertencentes, representados pelo elemento `<object>`, e as ligações entre objetos, representadas pelo elemento `<link>`.

O exemplo a seguir apresenta um trecho de um espaço de trabalho em XML (ilustrado na Figura 4-24). Nele existem dois objetos (`Corpo` e `Acionador`), ligados através de um elemento `<link>`, onde está definido que, quando o objeto `Acionador` recebe uma mensagem "clicado" (evento que indica que o objeto foi clicado), ele envia uma mensagem "ativa" para o objeto `Corpo`.

O uso de XML para representação dos espaços de trabalho os tornam bastante portáveis e adequados à Internet. A linguagem XML tem se estabelecido como padrão para troca de informações na Internet, especialmente na *Web*. Utilizar XML para representar o estado dos objetos é também uma recomendação de Frank Manola [MAN98], em seu estudo de um modelo de objeto para *Web*.

---

[3] ***Using Elements and Attributes*** – seção do OASIS - *The XML Cover Pages*, disponível no endereço: http://www.oasis-open.org/cover/elementsAndAttrs.html, em 15/02/2002.

```
<workspace id="LocalFisica3"
           type="application/x-casamagica">
                  :

    <object id="Corpo" class="Anima.Lancamento_Obliquo">
      <attribute name="x">10</attribute>
      <attribute name="y">300</attribute>
      <attribute name="camada">0</attribute>
                  :
    </object>

    <object id="Acionador" class="Anima.Imagem_Botao">
      <attribute name="x">65</attribute>
      <attribute name="y">0</attribute>
      <attribute name="camada">0</attribute>
      <attribute name="arquivo">[imagens]botao.gif</attribute>
    </object>

    <link source="Acionador" stimulus="clicado"
          category="SIMPLE" label="ativa" target="Corpo"/>
                  :

</workspace>
```

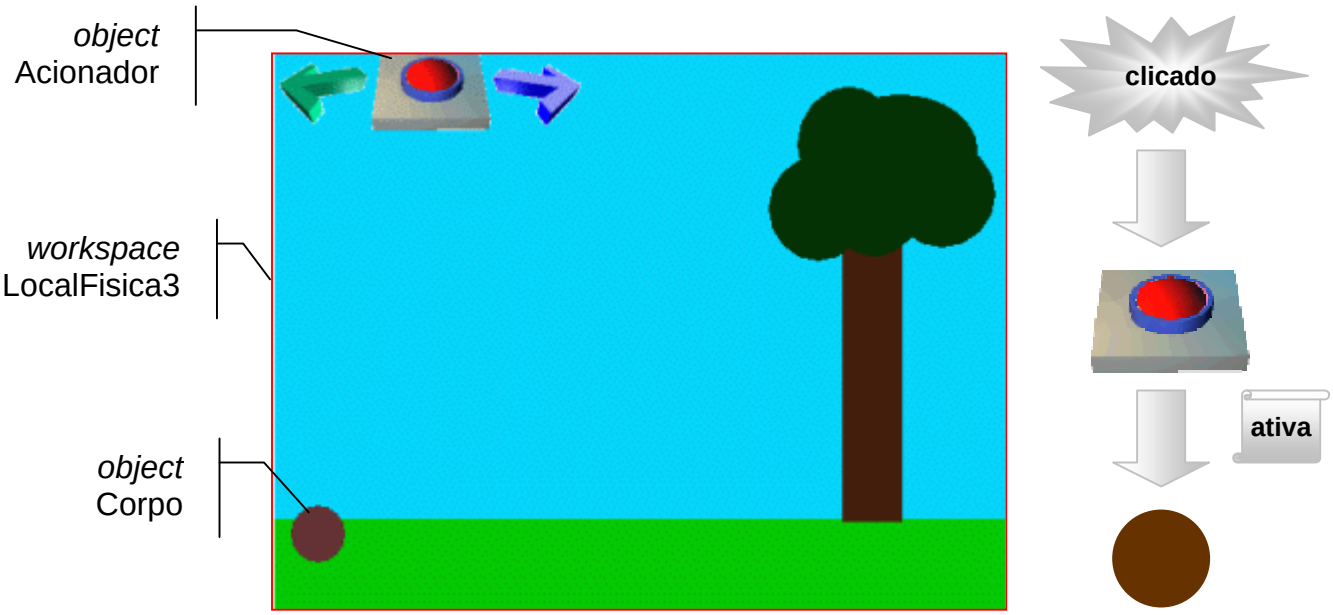


Figura 4-24: Ligação entre dois objetos em um espaço de trabalho.

Esta escolha de XML se estende para todas as representações do modelo *Anima*, como iremos observar nos tópicos seguintes.

4.3.2. Representação de Classe, Interface e Metadados (CIM) em RDF

Convencionalmente, em linguagens de programação, a ligação entre o comportamento de um objeto (definido por métodos) e o seu estado, é feita sob a forma de um ponteiro de memória que liga as variáveis de estado (atributos do objeto) aos métodos (reunidos em uma

classe). Deste modo: “A classe (em particular, os métodos que ela define) define o caminho pelo qual o estado poderá (e será) interpretado no sistema, e portanto é uma forma de metadado para o estado. Como resultado, a ligação entre um objeto e sua classe é essencialmente uma ligação de metadado” [MAN98].

A partir desta perspectiva das classes atuando como metadados dos objetos, Frank Manola sugere o uso de RDF – *Resource Description Framework* [BRI00] – para estabelecer a ligação entre o estado do objeto e os respectivos métodos que estabelecem seu comportamento.

RDF constitui uma tecnologia especializada na descrição de recursos através de metadados. Ela define mecanismos específicos para estabelecer a ligação entre um conjunto descritivo de metadados e o recurso descrito, o que a torna ideal para realizar a ponte entre o estado do objeto e seu comportamento, definido em uma linguagem de programação específica em um recurso independente.

Dentre as principais características de RDF, destacamos:

- Estabelece um modelo padronizado de associação de metadados a recursos. Isto vale inclusive para os meta-metadados, ou seja, metadados que descrevem outros metadados utilizados na descrição do recurso.
- Define esquemas para a descrição de recursos, com funcionalidades que incluem classes descritivas e mecanismos de herança para extensão e especialização de esquemas.
- É uma linguagem XML, portanto, se integra naturalmente com qualquer outro recurso XML.

Tudo isto torna RDF (Apêndice A.2) ideal para cumprir uma função que vai além do papel de ponte, atuando na descrição completa de classes, interfaces e metadados conforme o modelo CIM proposto.

A fim de utilizar RDF, mapeamos o CIM Schema, definido no Apêndice B.2, para um RDF Schema. O esquema completo, seus sub-esquemas, tipos e formatos definidos no apêndice, puderam ser mapeados para RDF (Apêndice C.3) sem perda de semântica, graças ao poder desta linguagem.

De um modo geral, cada esquema definido no Apêndice B.2 e os sub-esquemas definidos no Apêndice B.4 foram transformados em classes RDF. Seus elementos foram

mapeados para propriedades da classe. Tomemos como exemplo o sub-esquema External (Apêndice B.4):

External Schema						
Nome	**N.**	**Tipo**	**Formato**	**Descrição**	**Ob.**	**Comentários**
Identifier	1	string	Identifier	Identificador da associação.	S	Referência interna, utilizada localmente.
Reference	1	string	URI	URI associado ao rótulo.	S	Referência externa.

Ele foi mapeado para a seguinte classe RDF (Apêndice C.3):

```
<rdfs:Class rdf:ID="External">
  <rdfs:label>External association.</rdfs:label>
</rdfs:Class>

<rdf:Description rdf:about="#identifier">
  <rdfs:domain rdf:resource="#External"/>
</rdf:Description>

<rdf:Description rdf:about="#reference">
  <rdfs:domain rdf:resource="#External"/>
</rdf:Description>
```

Note que os elementos Identifier e Reference correspondem a propriedades da classe.

Tipos de dados e alguns formatos também foram mapeados para classes RDF. Os valores que podem ser assumidos por cada tipo, transformaram-se em instâncias das classes. Tomemos como exemplo o formato Visibility (Apêndice B):

Visibility	
public	Acessível por qualquer classe.
protected	Acessível dentro da classe ou classes herdeiras.
private	Acessível apenas dentro da classe.
package	Acessível por qualquer classe pertencente ao pacote.

Foi mapeado para classe RDF (Apêndice C.3):

```
<rdfs:Class rdf:ID="Visibility">
  <rdfs:label>Visibility.</rdfs:label>
</rdfs:Class>
```

Seus valores se transformaram em instâncias da classe:

```
<basic:Visibility rdf:ID="public">
```

```
  <rdf:value>public</rdf:value>
  <rdfs:label>Public</rdfs:label>
  <rdfs:comment>Visible to any class.</rdfs:comment>
</basic:Visibility>

<basic:Visibility rdf:ID="protected">
  <rdf:value>protected</rdf:value>
  <rdfs:label>Protected</rdfs:label>
  <rdfs:comment>Visible for class and their
subclasses.</rdfs:comment>
</basic:Visibility>

<basic:Visibility rdf:ID="private">
  <rdf:value>private</rdf:value>
  <rdfs:label>Private</rdfs:label>
  <rdfs:comment>Visible only for the class.</rdfs:comment>
</basic:Visibility>

<basic:Visibility rdf:ID="package">
  <rdf:value>package</rdf:value>
  <rdfs:label>Package</rdfs:label>
  <rdfs:comment>Visible for classes in the package.</rdfs:comment>
</basic:Visibility>
```

Todas as classes associadas a sub-esquemas e tipos convergem para o esquema principal, esquema CIM, que também foi mapeado para a classe CIM. Deste modo, uma descrição de uma classe conforme o esquema CIM, em RDF, será uma instância da classe CIM e deverá definir valores para as suas propriedades.

Em alguns experimentos informais de uso do esquema CIM em RDF, verificou-se que, apesar de seu grande poder de representação, RDF não é uma representação de fácil aprendizagem. Além disto, encontra-se em seus estágios iniciais de definição e aplicação, o que significa pouco material disponível para aprendizagem, poucos projetos práticos que explorem seu potencial e kits de desenvolvimento em estágios iniciais de produção e testes.

Todos estes fatores motivaram a criação de uma versão alternativa da representação CIM, mais simplificada, utilizando apenas XML com esquemas definidos a partir de uma DTD, tal como foi feito na representação da composição. A DTD está definida no Apêndice C.2. Do mesmo modo que a representação da composição, não foi possível estabelecer critérios precisos de mapeamento dos esquemas para a respectiva DTD.

Grande parte dos recursos oferecidos pelos esquemas RDF não possui equivalentes nos esquemas propiciados pela DTD, cuja função se limita a definir a estrutura de construção de documentos XML. Enquanto em RDF foram utilizadas classes para a representação de esquemas, na DTD foram utilizadas entidades. Por exemplo, o sub-esquema `External`, cujo

mapeamento em RDF foi apresentado anteriormente, é mapeado para as seguintes entidades em uma DTD:

```
<!ENTITY % external.element "EMPTY">
<!ENTITY % external.attlist
          "label      CDATA  #REQUIRED
           reference  CDATA  #REQUIRED">
```

Estas duas entidades ilustram o processo utilizado para o mapeamento dos esquemas para DTD XML. Alguns elementos do esquema são convertidos em atributos e outros em elementos, por este motivo, cada sub-esquema foi associado a duas entidades: as que possuem sufixo "`element`" definem a parte correspondente aos elementos e as de sufixo "`attlist`" definem a parte correspondente aos atributos.

Tipos de dados e alguns formatos foram mapeados para classes em RDF e seus valores são instâncias das mesmas. Na DTD, por outro lado, nos limitamos a definir cada um destes tipos e formatos como uma lista de possíveis valores. O exemplo apresentado em RDF, referente ao formato `Visibility`, é mapeado na DTD da seguinte maneira:

```
<!ENTITY % visibility.type "public|protected|private|package">
```

Está claro que esta representação alternativa implica uma significativa perda de semântica da representação resultante, quando comparada com a versão RDF. Por este motivo, adotamos em nosso trabalho a representação RDF e a implementamos em todas as ferramentas associadas a *Anima*.

A versão em XML sem RDF é apresentada como opção simplificada, para programadores que pretendam desenvolver ferramentas compatíveis com *Anima* e não querem adotar RDF, pelos motivos citados anteriormente.

Conforme será explicado na seção de implementação, a fim de minimizar o efeito da perda de semântica para aqueles que pretendem adotar o formato simplificado, construímos uma ferramenta que mantém a representação CIM em RDF e possui opção de exportá-la para uma outra versão XML sem RDF, apenas para simplificar o acesso por parte de outras aplicações.

4.3.3. Mecanismo de conversão utilizando XSLT

Conforme foi abordado no tópico 4.2.3, onde são descritos os mecanismos de conversão do modelo genérico, *Anima* adota uma metodologia de conversão baseada em programa genérico, que utiliza um plano de transformação para converter a composição

representada em um formato genérico em uma composição equivalente, implementada em uma linguagem (ou conjunto de linguagens), pronta para ser executada no computador.

XSL – *Extensible Stylesheet Language* [ADL01] – consiste em uma linguagem XML utilizada para elaboração de folhas de estilo, que possui uma linguagem, um sub-componente, denominada XSLT – XSL *Transformations* [CLA99] – cuja função é realizar a transformação de um documento XML em um outro documento XML, ou em alguns tipos específicos de saída, tais como um documento HTML ou um texto sem formatação.

Estas características tornam o XSLT ideal para a proposta de *Anima*. Sua metodologia de trabalho se baseia em um plano de transformação, representado por uma folha XSLT, que descreve todo o processo de conversão da origem para o destino. Um programa capaz de interpretar e executar o roteiro de uma folha de estilo (mecanismo de transformação), carrega o recurso (ou recursos) XML, a folha XSLT e gera a saída resultante da transformação. Retomando a Figura 4-17, o plano de transformação indicado no diagrama será, neste caso, a folha XSLT.

Existem várias modalidades de programas que cumprem o papel do mecanismo de transformação XSLT:

- Utilitários, instalados na máquina do usuário, que lêem os arquivos XML e XSLT, gerando a saída resultante, geralmente sob a forma de um outro arquivo.
- Bibliotecas acopladas a servidores *Web*, capazes de carregar arquivos XML e XSLT e enviar para a estação cliente apenas o resultado, geralmente representado por uma página HTML.
- Bibliotecas disponíveis publicamente em diversas linguagens de programação, para uso no desenvolvimento de sistemas que suportem XSLT.
- Navegadores *Web* capazes de realizar a transformação diretamente na máquina do cliente, no momento em que o navegador recebe a página XML e a respectiva página XSLT, apresentando ao usuário apenas o resultado da transformação.

Objeto	
Classe	Botao
Referência	X
Atributo	Valor
Rotulo	“Liga”
Largura	60
Altura	40

Objeto (XML)

```
<object id="x" class="Botao">
  <attribute name="Rotulo">"Liga"</attribute>
  <attribute name="Largura">60</attribute>
  <attribute name="Altura">40</attribute>
</object>
```

Plano de transformacão

```
{Classe} {Referência};                          (A)

{Referência} = novo {Classe};                   (B)

[Para cada Atributo]
  {Referência}.{Atributo} = {Valor};            (C)
[Fim]
```

Plano de transformação (XSLT)

```
<xsl:template match="object">
  <xsl:value-of select="@class"/>
  <xsl:text> </xsl:text>
  <xsl:value-of select="@id"/>
  <xsl:text>;                                   (A)
</xsl:text>

  <xsl:value-of select="@id"/>
  <xsl:text> = novo </xsl:text>
  <xsl:value-of select="@class"/>               (B)
  <xsl:text>;</xsl:text>

  <xsl:for-each select="attribute">
    <xsl:text>
</xsl:text>
    <xsl:value-of select="../@id"/>
    <xsl:text>.</xsl:text>
    <xsl:value-of select="@name"/>
    <xsl:text> = </xsl:text>                    (C)
    <xsl:value-of select="."/>
    <xsl:text>;</xsl:text>
  </xsl:for-each>

</xsl:template>
```

Produto

```
Botao X;

X = novo Botao;

X.Rotulo = “Liga”;

X.Largura = 60;

X.Altura = 40;
```

Figura 4-25: Associação dos elementos do Modelo Genérico à tecnologia XSLT.

Este conjunto de possibilidades torna XSLT (Apêndice A.3) uma solução ideal para um modelo adequado à *Web*, oferecendo diversas soluções alternativas para aplicação da folha de transformação.

A

Figura 4-25 apresenta uma especificação do plano de transformação (apresentados na Figura 4-18 do tópico 4.2.3) em XSLT. Através da figura podemos analisar a transformação da representação de um objeto (mapeada para XML), para um programa (hipotético)

resultante, a partir de um plano de transformação (especificado em XSLT), que efetua a substituição de campos.

Ao contrário dos outros tópicos, não apresentamos aqui um plano de transformação padrão. Isto porque este plano sofre variações, de acordo com a linguagem e ambiente destino. Deste modo, decidimos apresentar um plano de transformação exemplo, em XSLT, que ilustre a nossa proposta e sirva como referencial para outras implementações. Apresentamos no Apêndice C.4 a folha XSLT exemplo, que realiza a conversão de uma composição XML em uma página HTML, onde é disparada uma aplicação Casa Mágica (o sistema Casa Mágica possui um módulo *run-time* escrito na forma de uma *applet* Java).

4.3.4. Protocolo de comunicação entre objetos – Mensagens em XML

Na seção 4.2.4 foi mencionado que, a fim de superar as diferenças entre formatos específicos de diferentes sistemas e linguagens de programação, *Anima* transforma todo tipo de mensagem, trocada entre objetos de uma composição, em uma cadeia de caracteres.

A partir desta ótica, XML se mostra uma excelente opção de representação destas mensagens, pois:

- Por ser representada em formato texto, não está sujeita a incompatibilidades em plataformas diferentes.
- A adoção de XML mantém uma coerência com o restante do modelo. Isto é importante quando desejamos enviar, em conjunto com uma mensagem, a instância de um objeto. Podemos serializá-lo no mesmo formato (XML) da mensagem.
- É uma representação estruturada, bastante conveniente para a caracterização de tipos diferentes de mensagens e representação de estruturas complexas, como objetos.

Na representação XML, cada uma das partes da mensagem é mapeada da seguinte maneira:

- **Categoria**: Definida pelo nome do elemento que representa a mensagem, conforme mapeamento definido para cada categoria. As categorias disponíveis são: operação simples (sem parâmetro), operação de parâmetro único, modificação de propriedade, consulta de propriedade, valor de propriedade ou variável, operação

com múltiplos parâmetros ordenados, operação com múltiplos parâmetros nomeados.

- **Tipo**: Definido pelo atributo "type" do elemento. Se este atributo não aparecer, significa que a mensagem não é tipada.
- **Rótulo**: Definido pelo atributo "id" do elemento.
- **Parâmetros**: Sua configuração depende da representação da mensagem adotada.

Vamos exemplificar duas representações de mensagem em XML (Apêndice C.5), partindo da notação da tripla, introduzida na tabela "Formato dos Parâmetros conforme a Categoria", disponível no Apêndice B.3.

Tripla (Categoria/Tipo; Rótulo; Parâmetros)	Representação XML
(SINGLE; sum; 10)	`<M1 id="sum">10</M1>`
(SINGLE/amount; sum; 10)	`<M1 type="amount" id="sum">10</M1>`

Note-se que a categoria da mensagem (SINGLE) é mapeada para o rótulo do elemento (M1). O tipo da mensagem e o rótulo se transformam em atributos do elemento. O mapeamento dos parâmetros pode ser feito para atributos do elemento, como é apresentado no exemplo, ou para elementos.

Definimos dois mecanismos para se estabelecer o tipo da mensagem, quando se trata de uma mensagem tipada, ou seja, uma mensagem que segue um esquema de construção.

Através do XML Schema [DOD01] é possível descrever detalhadamente a estrutura de um recurso XML, neste caso, a mensagem representada. Além disto, XML Schema define tipos de dados primitivos, adequados para caracterizar os parâmetros associados a este tipo de dados na mensagem.

O segundo mecanismo utiliza descrições RDF. Se, por um lado, RDF não define de forma nativa tipos de dados primitivos, e não é especializado na declaração da estrutura do recurso XML[4], por outro, oferece meios para a estruturação de uma descrição, com recursos semânticos superiores ao XML *Schema*.

A combinação de um XML Schema com uma descrição em RDF estabelece uma infra-estrutura ideal para a caracterização de um tipo de mensagem: enquanto o XML Schema detalha aspectos da estrutura da mensagem, RDF atua na semântica da mensagem, definindo

[4] Apesar do RDF Schema definir a estrutura de uma descrição RDF, que em si é feita em XML, ela não define a estrutura de um recurso XML qualquer (que não seja RDF), tal como a representação da mensagem *Anima*.

metadados descritivos e uma estrutura de classificação – utilizando inclusive sua hierarquia de classes – que é de grande importância para definir, por exemplo, a compatibilidade entre mensagens, no momento da construção da composição.

Este tópico conclui o delineamento da representação *Anima* baseada em marcação estruturada. Iremos agora descrever a infra-estrutura utilizada na implementação do modelo proposto.

4.3.5. Aplicando a representação baseada em Marcação Estruturada

Este tópico sintetiza, através de um exemplo, a aplicação de *Anima*, destacando-se seus três aspectos fundamentais: representação da composição, mecanismos de conversão e protocolo de comunicação. Para isto, estendemos um exemplo, inicialmente apresentado no tópico 4.2.1.1, ilustrado na Figura 4-5, que trata de uma composição integrando objetos desenvolvidos em duas linguagens de programação diferentes (Casa Mágica e Alice), em um navegador *Web*.

No exemplo, um professor de física deseja fazer uma simulação de uma esfera, que é lançada obliquamente a partir do solo. Ele pretende apresentar simultaneamente: um gráfico que trace em um plano bidimensional o trajeto percorrido pela esfera e o movimento desta mesma esfera em um ambiente tridimensional.

Isso foi resolvido através da criação de um documento *Web*, que integra uma produção feita em Casa Mágica com outra em Alice. Tal documento *Web* consiste em uma página HTML onde estão inseridos objetos estrangeiros: um que apresenta o lançamento em um plano bidimensional, construído através do sistema Casa Mágica e outro que apresenta o mesmo lançamento no espaço tridimensional, construído através do sistema Alice.

Inicialmente, é elaborada a composição com representação em XML. Para obter um documento *Web* com objetos estrangeiros, a partir da representação da composição, utiliza-se o mecanismo de conversão baseado em folhas de estilo XSL. O processo completo é ilustrado na Figura 4-26.

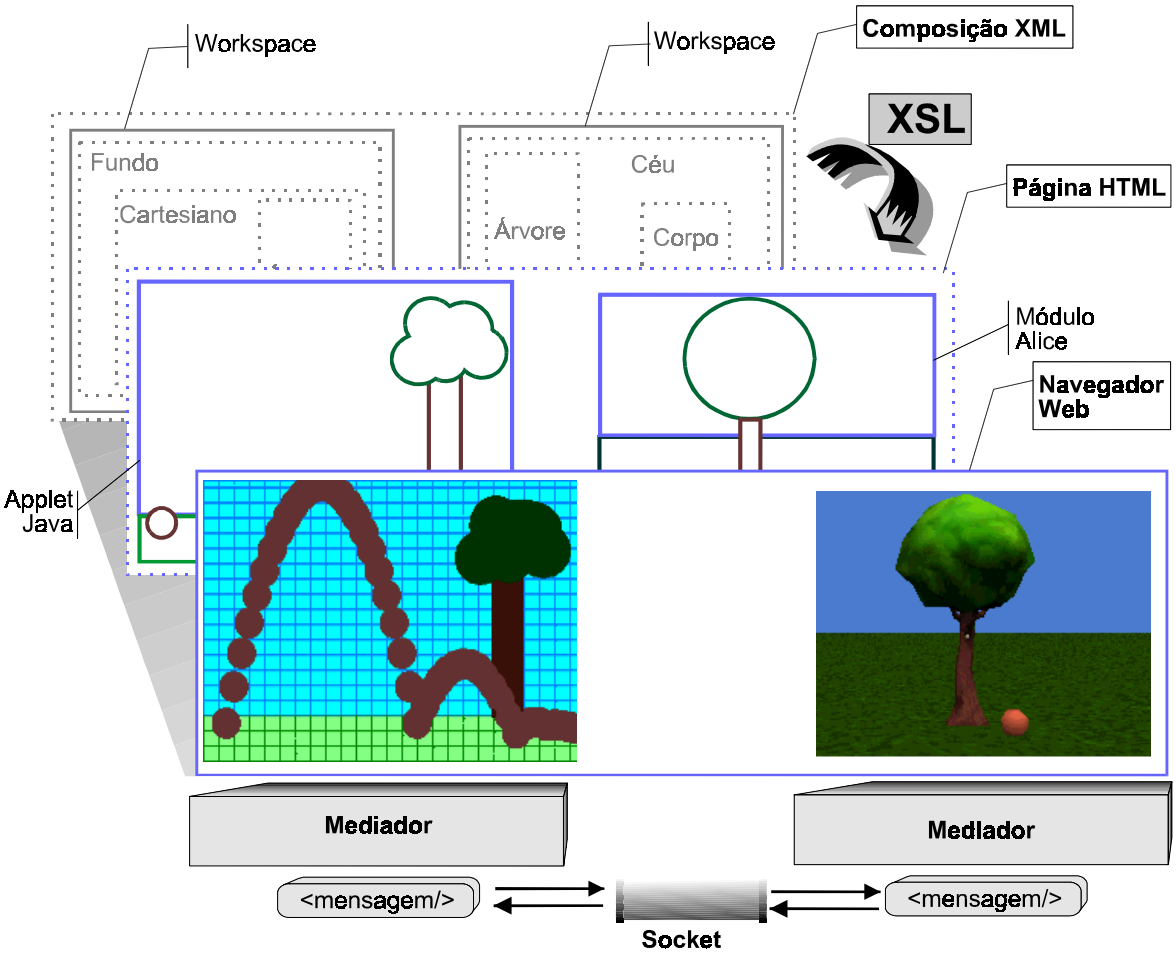


Figura 4-26: Diagrama ilustrando o processo de transformação, que parte da representação XML até o documento apresentado no navegador *Web*.

Neste exemplo, utiliza-se uma folha de estilo para converter o espaço de trabalho Casa Mágica em uma chamada da *applet* Java com seu *runtime*, como também para converter o espaço de trabalho Alice para uma chamada de seu *plugin*.

A comunicação entre os objetos Casa Mágica e Alice se faz através de seus respectivos Mediadores escritos em suas linguagens nativas. Como está ilustrado na Figura Figura 4-26, eles são conectados através de um *socket* TCP/IP (por se tratar de um documento *Web*, é certo que haverá disponibilidade deste protocolo na máquina onde está sendo apresentado o documento). Os *sockets* são uma maneira versátil de transferir seqüências de caracteres entre aplicações diferentes.

4.3.6. Utilizando *Anima*

Este tópico apresenta os procedimentos necessários para se construir uma aplicação que utilize o modelo *Anima*, ou para adaptar uma aplicação já existente.

Em primeiro lugar, é importante destacar que nem sempre será necessário implementar todo o modelo. Aplicações podem fazer uso de apenas alguns aspectos de *Anima*, ou implementar de forma gradativa cada uma das funcionalidades. De qualquer modo, existe uma seqüência recomendada de implementação, dado que alguns aspectos dependem de outros. Apresentaremos a seguir o modo como implementar cada um dos aspectos separadamente, e depois, a seqüência recomendada para fazê-lo.

A. Protocolo padronizado de comunicação

O Mediador é peça chave para a execução da comunicação; caso não exista um implementado na linguagem em questão, será necessário codificá-lo um. No tópico 4.2.4 são detalhadas as funções que desempenha um Mediador. Ele deve possuir um conjunto de interfaces padronizadas para a interação com os objetos *Anima*, principalmente a `AnimaInterface`. Esta interface possui uma versão implementada em Java e pode servir como referencial para outras linguagens.

Uma vez que exista um Mediador implementado, o próximo passo é a construção de classes capazes de enviar e receber mensagens em formato XML, conforme esquema definido por *Anima*. A troca destas mensagens pode acontecer de três formas, como é ilustrado na Figura 4-20 e apresentado no tópico 4.2.4.

As classes trocam suas mensagens XML com o Mediador através de um único método (`taggedMessage`), deste modo, na interface de ambos, este método sempre estará disponível (isto é definido na interface `AnimaInterface`). Todo o conteúdo que acompanha a mensagem é enviado dentro dela em XML e cada mensagem alcança seu destino através do Mediador.

Se a linguagem já possuir um kit de suporte para *Anima*, como acontece em Java, a construção e desmontagem das mensagens em XML podem ser feitas com auxílio de bibliotecas de suporte.

As bibliotecas de suporte incluem recursos para serialização e deserialização de objetos, caso seja necessários incluí-los em uma mensagem.

Sintetizando:

Para implementar suporte a Anima em uma nova linguagem de programação é necessário:

- codificação de um Mediador;
- definição das interfaces *Anima* para a nova linguagem;
- construção de bibliotecas de suporte para mensagens e serialização e deserialização de objetos.

Para que uma classe possa enviar e receber mensagens segundo o protocolo definido por *Anima*, ela precisa implementar a respectiva interface e ser capaz de montar e desmontar mensagens em XML.

B. Representação da Composição

Para ferramentas que desejam ler ou gerar composições *Anima* é necessário, em primeiro lugar, construir ou adaptar as classes dos objetos que estarão envolvidos na composição, para que possam se comunicar conforme o protocolo de mensagens definido no item anterior.

A composição propriamente dita, está registrada em um arquivo XML, o qual precisará ser lido e/ou gravado pela ferramenta.

A biblioteca, já implementada em Java, possui suporte para leitura de composições *Anima*, traduzindo-as para um modelo de objetos interno, que reflete a estrutura da composição. Ela também permite a gravação deste modelo em XML. Bibliotecas equivalentes podem ser estendidas para outras linguagens.

C. Representação CIM

Tal como acontece para a Representação da Composição, para acessar a representação CIM, a ferramenta deve realizar leitura e gravação em arquivo XML, que contém os metadados RDF.

A biblioteca já implementada em Java também possui suporte para leitura e gravação de arquivos CIM/RDF. Ela utiliza um modelo interno para representação CIM, com facilidades de edição, acesso a informações da classe e metadados. Adicionalmente, possui um suporte à coleta de dados CIM, a partir do mecanismo de reflexão em classes Java. Também neste caso, bibliotecas equivalentes podem ser estendidas para outras linguagens.

D. Mecanismos de Conversão

Em princípio, os mecanismos de conversão XSL, construídos para uma linguagem de programação como Java, podem ser utilizados para qualquer composição *Anima* cujos objetos envolvidos estejam em Java.

Novas folhas XSL podem ser construídas para linguagem de programação que ainda não possua o mecanismo de conversão implementado, ou quando se deseja produzir um resultado em formato diverso daqueles já existentes. Por exemplo, adaptar uma folha XSL que gera código Java a ser usado em uma aplicação independente, para que gere uma *applet* dentro de uma página.*Web*.

4.4. Ferramentas

O modelo *Anima* proposto, representado em XML, RDF e XSL, foi implementado em um sistema construído sobre a plataforma Java.

Esta implementação não se restringiu a definir soluções práticas e imediatas para o sistema alvo, mas foi construída sobre um conjunto de bibliotecas e ferramentas projetadas para serem utilizadas não apenas neste sistema, como também em qualquer outro que faça uso de *Anima*.

Deste modo a implementação foi dividida em três partes:

- **Bibliotecas** – foram elaboradas bibliotecas abertas e genéricas, capazes de desempenhar as principais tarefas relacionadas à leitura, gravação e ao processamento dos dados associados ao modelo e à representação *Anima*. Elas cumprem três papéis importantes: estabelecem uma implementação referencial do modelo; são utilizadas pelas ferramentas e pelo sistema, elaborados neste projeto; e podem ser utilizadas por outras ferramentas e sistemas que façam uso de *Anima*.

- **Utilitários CIM** – este conjunto de ferramentas tem a capacidade de capturar representações internas de classes Java; possui uma ferramenta de criação e edição de dados CIM (*Anima Workshop*) e permite a leitura e gravação da representação em RDF ou XML.

- **Autoria e execução de composições** – o módulo responsável pela autoria de composições foi construído dentro do ambiente Casa Mágica, cuja estrutura já era baseada em objetos de *software* e foi adaptada para o padrão *Anima*. As

composições resultantes podem ser executadas no próprio ambiente ou em um navegador *Web*, utilizando um módulo *run-time*.

A seguir, apresentaremos cada uma das partes, separadamente.

4.4.1. Bibliotecas

Podemos categorizar as bibliotecas criadas, conforme os seguintes grupos:

- **Suporte a objetos Anima** – classes utilizadas por objetos Anima, a fim de se adequarem à edição e execução de composições conforme este padrão.
- **Acesso e manutenção de representação CIM** – classes que fornecem acesso a representações CIM e dão suporte à sua manutenção.

As classes mais relevantes de cada uma das bibliotecas estão relacionadas no Apêndice D.1.

4.4.2. Utilitários CIM

Para cada uma das tarefas mais comuns de manutenção e acesso a uma representação CIM, foram criados pequenos utilitários independentes em Java. Conforme esquematizado na Figura 4-27, estes pequenos utilitários desempenham tarefas em torno de uma representação comum em RDF.

Figura 4-27: Relação entre utilitários e Representação CIM em RDF.

Novos utilitários podem ser criados para a realização de tarefas parciais, integrando-se com os demais através da representação CIM. O utilitário `ExtractRDF`, por exemplo, realiza a extração das características de um JavaBean e gera um arquivo RDF. Para extrair a representação CIM em RDF de uma classe escrita em outra linguagem, um utilitário `ExtractRDF` pode ser escrito na respectiva linguagem. Como a representação RDF é independente de plataforma e linguagem, o resultado se integrará naturalmente com os demais utilitários.

Os principais utilitários, conforme descrição acima, estão relacionados no Apêndice D.2.

Além dos pequenos utilitários, foi criada uma ferramenta integrada para suporte à manutenção de uma representação CIM, denominada Estúdio *Anima*. Esta ferramenta implementa as três primeiras etapas da metodologia para produção, manutenção e uso de CIM, conforme descrito no tópico 4.2.2 "Representação de Classe, Interface e Metadados (CIM)":

a) importação da estrutura da classe por reflexão (ilustrado na Figura 4-10);

b) modelagem e edição da representação (ilustrado na Figura 4-11);

c) armazenamento da representação (ilustrado na Figura 4-12 e na Figura 4-13).

A Figura 4-13 ilustra uma interessante aplicação de integração entre pequenos utilitários independentes e o Estúdio *Anima*. A extração dos dados e geração do arquivo CIM/RDF, conforme descrito anteriormente, pode ser feita por um utilitário escrito em qualquer linguagem de programação e editada pelo Estúdio *Anima* em Java.

4.4.3. Autoria e execução de composições

Conforme já foi abordado anteriormente, a autoria das composições foi implementada dentro do ambiente Casa Mágica.

Para isto, o ambiente teve que sofrer as seguintes adaptações:

- A representação dos componentes de *software*, utilizados no ambiente de produção, são carregadas a partir de uma representação CIM em RDF.
- O ambiente de construção das composições oferece suporte a objetos Java e Casa Mágica, conforme especificação *Anima*.
- O ambiente permite inserir na composição objetos definidos em outras linguagens, apresentando no ambiente de autoria apenas a sua fotografia (definida na representação CIM).
- A partir das interfaces, definidas em CIM, o ambiente estabelece quais atributos/propriedades de um objeto que podem ser apresentados e editados, além das mensagens que ele é capaz de enviar e eventos que é capaz de responder.
- A composição é armazenada em XML, conforme o esquema definido por *Anima*.

- Os objetos trocam mensagens em formato XML, conforme protocolo definido por *Anima*.
- Casa Mágica implementa um mediador *Anima*, conforme definido no tópico 4.2.4 “Protocolo padronizado de comunicação”, utilizado na execução da composição no próprio ambiente e em módulo *run-time* para navegador *Web*.

Casa Mágica implementa as duas últimas etapas da metodologia para produção, manutenção e uso de CIM, conforme descrito no tópico 4.2.2 “Representação de Classe, Interface e Metadados (CIM)”:

d) composição baseada em CIM (ilustrada na Figura 4-14 e na Figura 4-15);

e) armazenamento da composição (ilustrado na Figura 4-16).

No próximo capítulo apresentaremos um projeto desenvolvido, utilizando *Anima* dentro do ambiente Casa Mágica. Este estudo de caso propicia uma visão mais abrangente de como todos os elementos *Anima* se combinam neste ambiente, e permite melhor ilustrar suas facilidades.

5. PROJETOS EDUCACIONAIS COM ANIMA

Neste capítulo realizamos um estudo referente à aplicação do modelo e das ferramentas proposta com o objetivo de: ilustrar possíveis aplicações de *Anima*, verificar de forma concreta as propostas do modelo e fornecer subsídios para uma análise do projeto e um confronto com projetos equivalentes. O capítulo também sintetiza as principais contribuições de *Anima* no seu universo de atuação.

Foi escolhido um estudo de caso para realizar esta análise, em que se explora as principais funcionalidades de *Anima*, ilustrando uma categoria de projetos educacionais para os quais o modelo foi concebido.

No tópico – **Máquinas e componentes de *software* na aprendizagem matemática** – é apresentado um projeto piloto, que consiste em uma biblioteca de componentes de *software* capazes de realizar operações matemáticas elementares. A biblioteca deve ser acoplada à ferramenta de autoria apresentada no capítulo anterior, e pode ser usada tanto pelo professor, para elaborar uma demonstração animada do comportamento de um fenômeno matemático, como pelo aluno, para modificar uma aplicação construída pelo professor ou criar suas próprias aplicações. As produções podem ser elaboradas localmente e distribuídas através da Internet, ou ser apresentadas diretamente em páginas *Web*. O tópico detalha aspectos da constituição dos componentes, conforme o modelo *Anima*, e benefícios de seu uso neste contexto.

Em **Casa Mágica e *Anima***, é discutido o papel de *Anima* dentro do projeto prático construído no tópico anterior (em Casa Mágica) e é realizada uma síntese das principais contribuições, observadas tanto no domínio específico do uso da informática na educação, como também em um domínio mais abrangente, que envolve a integração da tecnologia de objetos com a Internet.

5.1. Máquinas e componentes de *software* na aprendizagem matemática

Alguns tipos de atividades pedagógicas têm o papel de conduzir o aprendiz à concepção de uma “mecânica” regular e/ou previsível que existe por detrás de um objeto de estudo. Isto é especialmente interessante em áreas de conhecimento, como a matemática, que se deparam freqüentemente com tais objetos.

No tópico 2.2.1 foi explorada a importância na escolha da metáfora como mediadora entre o indivíduo e a máquina/sistema. Também no ensino de matemática a metáfora cumpre um papel igualmente importante.

Nilson José Machado, ao analisar o papel da metáfora no ensino da matemática, enfatiza sua importância como uma ponte entre dois contextos: “Ora, é precisamente no estabelecimento de pontes entre diferentes contextos, na iluminação de relações estruturais que subjazem, a despeito da diversidade dos campos semânticos, que a metáfora afigura-se como instrumento fundamental” [MAC01].

No contexto da matemática a metáfora das máquinas é especialmente interessante. As máquinas são artefatos cujo comportamento se insere em um universo tipicamente mecânico. A maioria delas possui um funcionamento bastante previsível.

O uso da metáfora das máquinas, como mediadora na concepção do comportamento de certos fenômenos matemáticos, por si só é uma poderosa ferramenta para o desenvolvimento de atividades de ensino-aprendizagem.

Elas são bastante usadas no ensino da matemática, por exemplo, para a explicação das quatro operações básicas e, mais adiante, para explanar o funcionamento das funções matemáticas.

Para explicar as quatro operações básicas, as máquinas são convencionalmente apresentadas como na Figura 5-1, como um artefato que recebe dois números de entrada e gera um número resultante na saída.

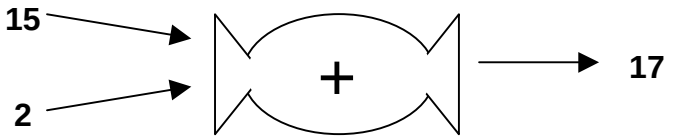


Figura 5-1: Representação de uma máquina de soma.

Em 1996, o autor deste trabalho desenvolveu um programa que explora a metáfora das máquinas, para o desenvolvimento de exercícios que envolvem somas e subtrações de números inteiros. O programa foi aplicado a partir deste ano com alunos da primeira série do ensino fundamental, no Colégio Anchieta, Salvador-BA. A forma de apresentar as máquinas e o modo como elas executam as operações se baseou em uma linguagem e notação utilizadas pelos professores do colégio com seus alunos.

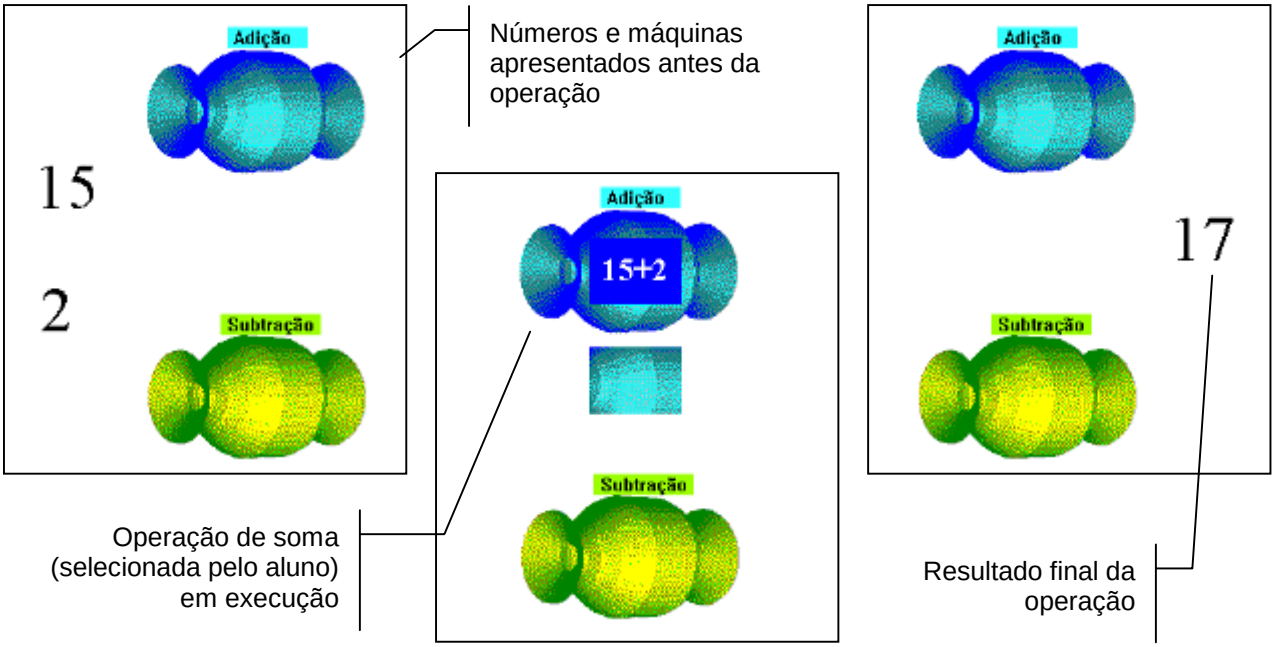


Figura 5-2: Programa que utiliza a metáfora das máquinas para exercícios de soma e subtração.

A Figura 5-2 ilustra três momentos do programa, apresentando visualmente a execução de uma soma. Neste programa, o aluno interage com as máquinas e deve solucionar desafios propostos pelo sistema, escolhendo as máquinas adequadas para cada situação.

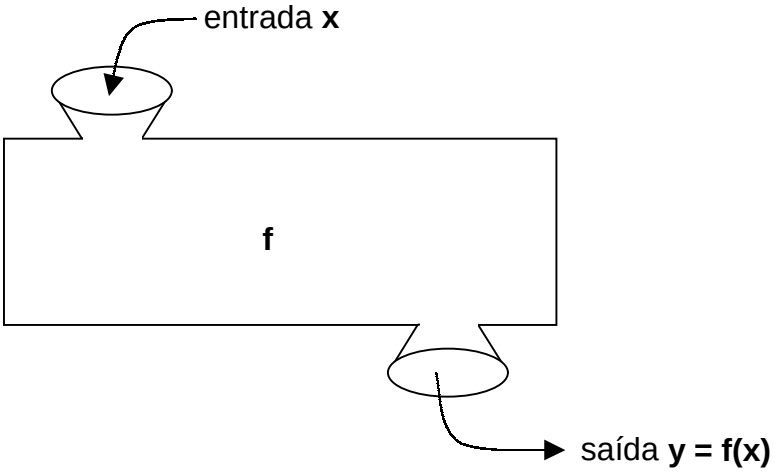


Figura 5-3: Ilustração do comportamento de uma função através de uma máquina.

As funções matemáticas são apresentadas em forma de máquinas, como a ilustrada na Figura 5-3. A “máquina função” (`f`) é tomada em um único bloco, que recebe como entrada um valor (`x`) e retorna um valor resultante (`y`), calculado a partir da entrada (`f(x)`).

A Figura 5-4 apresenta um uso bastante interessante da metáfora das máquinas funções, estabelecendo um paralelo com animais, utilizado pelo *site Natural Math*[1]. Nesta máquina, a operação aplicada sobre o animal consiste no retorno à sua infância.

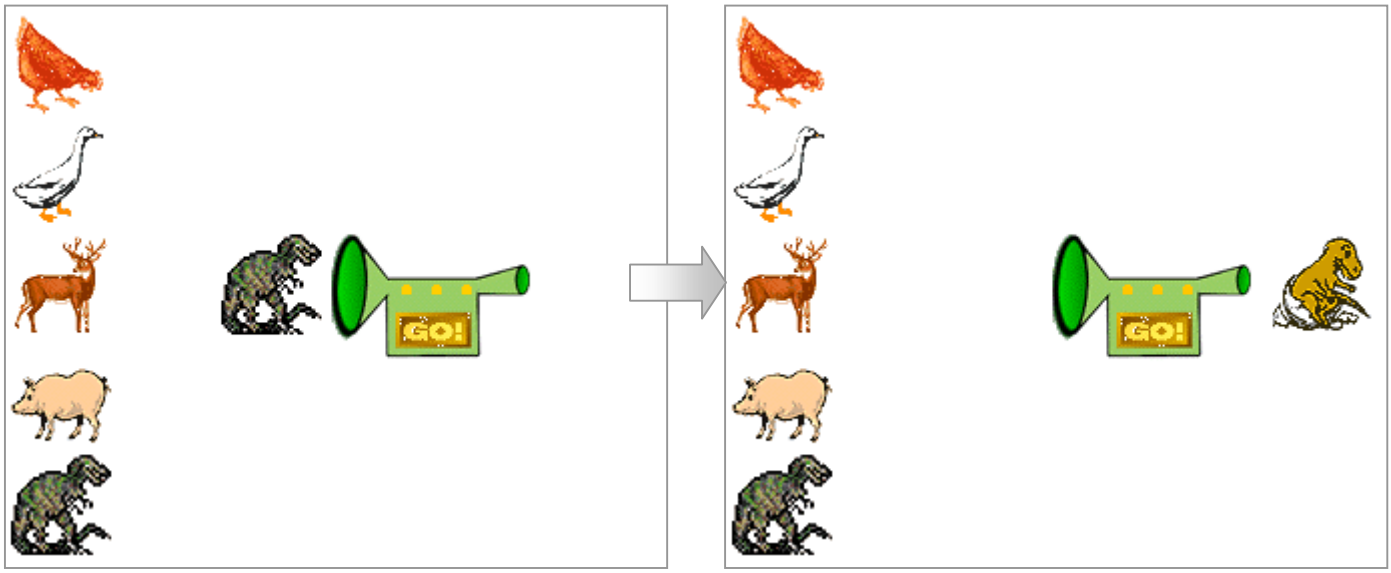


Figura 5-4: Máquina-função ilustrada com animais.

A proposta deste estudo de caso consiste em realizar uma mixagem da abordagem utilizada no ensino fundamental, onde operações mais simples são representadas sob a forma de máquinas, com a abordagem utilizada para o ensino das funções, cujas máquinas representam operações mais complexas. Iremos estender o uso da metáfora das máquinas para a representação interna do funcionamento da função. Deste modo, cada elemento, responsável por um aspecto da máquina função, pode ser analisado individualmente em pequenas máquinas, utilizando a mesma metáfora da máquina do nível mais alto.

O computador, tomado como máquina genérica, é capaz de simular virtualmente estas máquinas matemáticas, cujo comportamento se pretende estudar. Deste modo, as máquinas mais elementares foram convertidas em componentes de *software*, compondo uma biblioteca do sistema Casa Mágica, utilizada por professores e alunos para a construção de máquinas funções, através da interligação das máquinas de menor complexidade.

---

[1] O endereço do *Natural Math* é http://www.naturalmath.com/ e a página do exemplo apresentado (máquina função) é http://www.naturalmath.com/functions.html.

Vamos apresentar como isto é feito, tomando para exemplo o estudo do comportamento de uma função de primeiro grau, tal como: **f(x) = 2x + 3**.

A primeira etapa para elaborar esta representação interna é detalhar a mecânica do que desejamos ilustrar. Tomemos, então, o seguinte detalhamento:

Para cada valor que a variável **x** pode assumir, a função será exatamente seu dobro mais três unidades. Deste modo, se tomamos um ponto descrito pelo par (x, f(x)) e o deslocamos pelo gráfico que descreve a função, para cada unidade de variação **x**, observamos que **f(x)** varia exatamente duas vezes seu valor. As três unidades que lhe são acrescidas (+3), por outro lado, não variam e sempre representam um acréscimo constante.

Em seguida, definimos cada pequena atividade executada na função sob a forma de uma máquina:

×	**Multiplicação:** realizada entre 2 e x.
+	**Soma:** realizada entre o resultado de 2x e 3.

As máquinas menores se interligam, de tal forma que o resultado de uma pode ser utilizado como entrada para outra. A Figura 5-5 ilustra este tipo de composição.

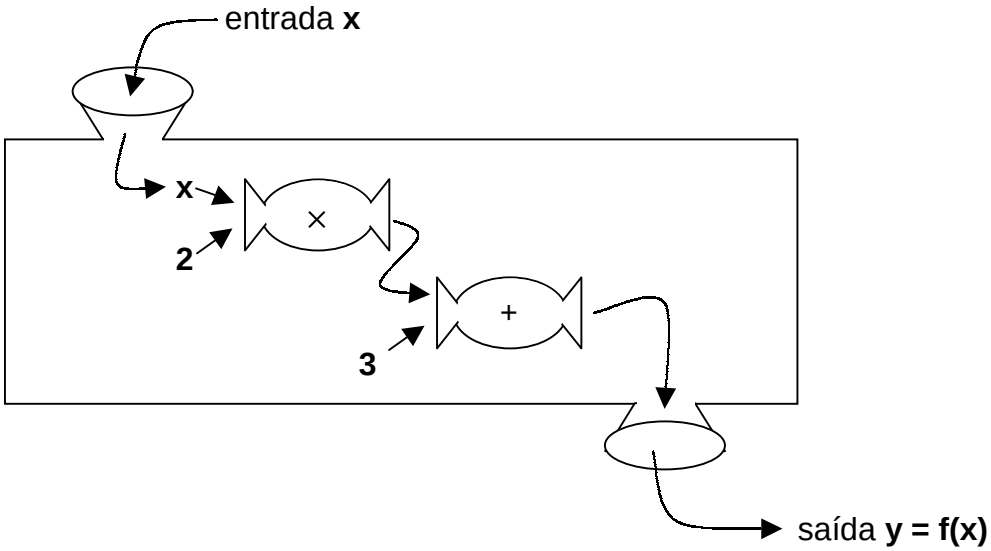
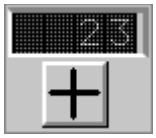


Figura 5-5: Máquina função composta por máquinas menores.

Conforme foi exposto, cada máquina é representada neste projeto por um componente de *software*, deste modo definimos os seguintes componentes:

	Multiplicação: realiza a multiplicação entre dois números disponíveis nas propriedades `OperandA` e `OperandB` (após a inserção de ambos), gerando o evento "action", que possui o resultado da operação agregado. Apresenta visualmente o resultado da operação em seu painel.
	**Soma**: realiza a soma entre dois números disponíveis nas propriedades `OperandA` e `OperandB` (após a inserção de ambos), gerando o evento "action", que possui o resultado da operação agregado. Apresenta visualmente o resultado da operação em seu painel.

Cada um dos componentes foi construído em Java, implementando as interfaces *Anima*, e para cada um deles foi elaborada uma representação CIM em RDF. Dentre suas características funcionais, é importante apresentar o mecanismo pelo qual eles se comunicam com os demais componentes e realizam sua função.

Ambos os componentes possuem duas propriedades associadas a atributos, que registram o valor de seus operandos, denominados `OperandA` e `OperandB`. O valor destas propriedades pode ser alterado através de mensagens da categoria WRITE, destinadas a modificar o valor de uma propriedade do objeto que a recebe. Neste caso, a mensagem:

Tripla	XML
(WRITE; OperandA; 2)	`<MW id="OperandA" v="2"/>`

modifica o valor da propriedade `OperandA` para o valor 2.

De posse do valor dos dois operandos, o componente realiza a soma e gera um evento "action"[2], que indica que a operação foi completada. Os eventos podem possuir valores agregados na forma de parâmetros, pois possuem a mesma estrutura das mensagens (em *Anima*, um evento é a notificação de um acontecimento na forma de uma mensagem).

O evento "action" destes componentes se apresenta da seguinte maneira:

Tripla	XML
(SINGLE; action; 20)	`<M1 id="action" v="20"/>`

Como pode ser observado, a mensagem é da categoria SINGLE, o que indica que ela vem acompanhada de um único valor, neste caso, o resultado da operação, que é o número 20.

Através de um *link* registrado na composição, este objeto pode ser ligado a um outro. Por exemplo, se a identificação deste objeto da classe Multiplicação for `ObjMultiplica` e

[2] O evento "action" sempre está associado à execução de uma ação típica do objeto, neste caso a realização da operação.

ele quiser enviar o resultado de sua operação para um objeto da classe Soma, cuja identificação seja `ObjSoma`, teremos:

```
<link source="ObjMultiplica" stimulus="action"
 category="WRITE" label="OperandA" target="ObjSoma"/>
```

Note que o evento “action” é considerado o estímulo que aciona o *link*. O evento estímulo possui um valor associado, que é automaticamente redirecionado para a nova mensagem. Esta, por sua vez, indica o registro do valor na propriedade `OperandA` de `ObjSoma`. Deste modo, ele enviará para o `ObjSoma` a mensagem:

Tripla	XML
(WRITE; OperandA; 2)	`<MW id="OperandA" v="20"/>`

Através deste processo, o resultado da multiplicação torna-se um operando da soma.

Além dos componentes apresentados, para completar a representação da função f(x)=2x+3, utilizando componentes de *software*, são necessários dois componentes adicionais.

	Entrada numérica: recebe um número inteiro pelo teclado e, ao ser clicado, o botão `Ok` gera um evento “action” com o valor digitado agregado.
	Constante numérica: neste componente fica registrado um valor inteiro constante na sua propriedade `Value`. O valor da constante é apresentado visualmente no painel.

A Figura 5-6 ilustra a composição completa, com todos os componentes citados, indicando cada mensagem trocada entre eles.

O componente de entrada numérica dispara todo o processo. Ele está ligado por *links* a três componentes: multiplicação, constante-2 e constante-3. O estímulo dos *links* é o evento “action”, que indica um novo valor disponível.

Para o componente de multiplicação, ele envia uma mensagem da categoria WRITE, registrando seu valor no `OperandA`. Para os dois componentes de constante numérica, ele envia uma mensagem simples, sem valor associado.

Ambas as constantes possuem *links* de modo que, ao receberem a mensagem do componente de entrada (este é o estímulo), enviam uma mensagem, registrando seus valores na propriedade `OperandB`, dos componentes de operação a que estão ligados (conforme indicado na figura).

O componente de multiplicação possui um *link* associado ao seu evento “action”, de modo que, ao terminar a operação, envia uma mensagem com o resultado para o componente de soma. Esta mensagem, da categoria WRITE, registra o valor do resultado da multiplicação na propriedade `OperandA` do componente de soma.

O componente de soma é o último em toda a seqüência. Ao receber os valores de seus dois operandos, executa a soma e apresenta-a em seu painel.

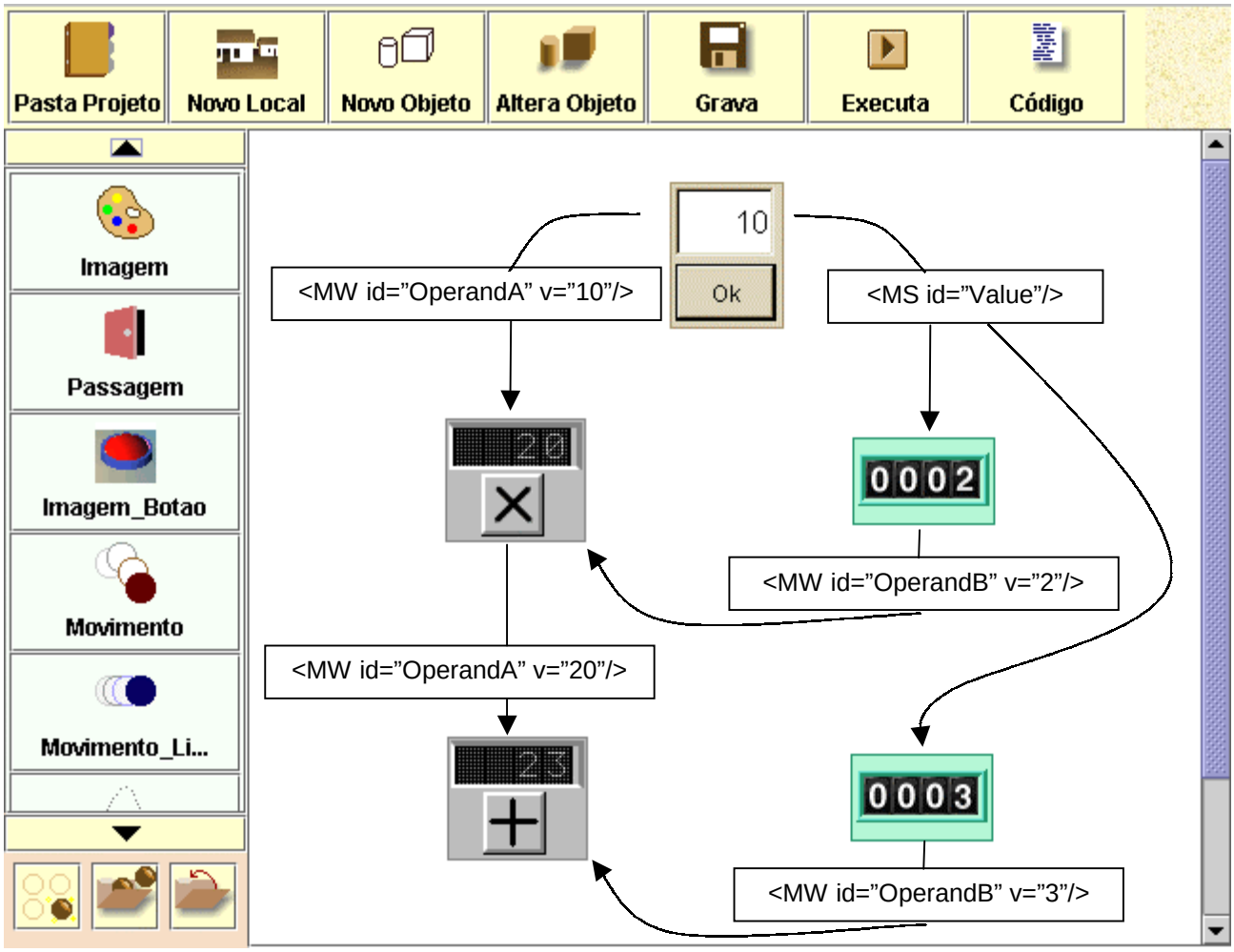


Figura 5-6: Composição em Casa Mágica que apresenta a função y=2x+3.

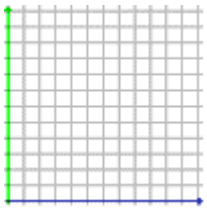
A estratégia adotada nesta composição conduz a uma exploração visual do funcionamento da função. No entanto, ela se restringe à execução do processo para cada valor isolado inserido pelo usuário.

Quando se estuda a construção de gráficos de função, é interessante a possibilidade de se realizar uma variação temporal no valor da variável de entrada (x), de modo que se possa observar como se comporta a curva da função calculada.

Por este motivo, elaboramos uma segunda composição capaz de explorar esta mesma função do ponto de vista dinâmico, ou seja, capaz de apresentar graficamente o modo como a função se comporta, na medida que **x** se desloca pelos valores do domínio.

Apesar do valor de **x** variar de forma contínua pelo domínio da função, para efeito didático tomaremos uma faixa de valores discretos.

Nesta segunda composição, adotamos uma abordagem diferente da anterior, por nos parecer uma representação mais adequada ao fenômeno explorado. Para isto, acrescentamos os seguintes componentes:

	Botão: ao ser clicado gera o evento “action”, que, através de um *link*, pode disparar o envio de mensagens para outros componentes.
	Relógio: gera o evento “action” em ciclos regulares. O tempo entre um ciclo e outro é configurado através da propriedade `delay`. Através de um *link* pode enviar mensagens em tempos regulares para outros componentes.
	Contador Variável: cada vez que recebe a mensagem “increment”, incrementa o valor em seu contador (propriedade de nome `value`), atualiza o valor do contador apresentado no painel e gera um evento “action” com o valor do contador agregado.
	Multiplicador: multiplica as mensagens recebidas segundo o valor da propriedade `factor` (indicada no painel), ou seja, se `factor` é dois, para cada mensagem recebida, o componente gera dois eventos “`action`”. Os eventos multiplicados podem ser redirecionados para outros objetos, na forma de mensagens multiplicadas (usando *links*).
	Par Ordenado: possui duas propriedades denominadas **x** e **y**. Quando ambas recebem um valor, o componente gera um evento “action” com os valores do par agregados.
	**Plano Cartesiano:** traça um gráfico no plano cartesiano interligando pontos, informados através de mensagens “insert point”. A cada mensagem recebida, atualiza o gráfico na tela.

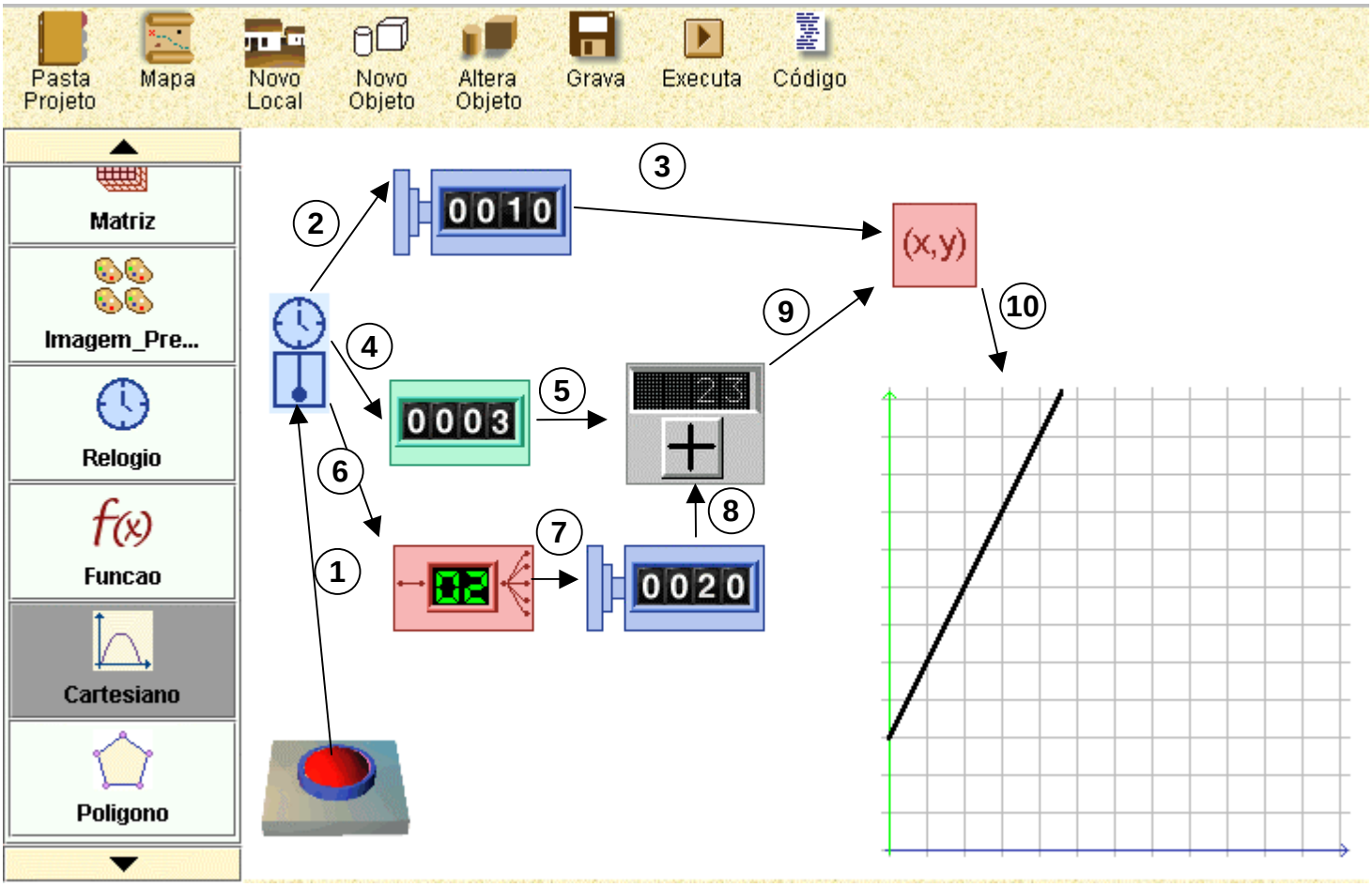


Figura 5-7: Outra composição em Casa Mágica que apresenta a função y=2x+3.

No exemplo apresentado, ao ser clicado o **botão**, produz-se um evento "action", que ativa o envio da mensagem "start" (através de um *link*), para o componente **relógio** (1). O relógio inicia seu trabalho de gerar eventos ("action") em tempos regulares. Estes eventos são encaminhados na forma de mensagens, por três *links*, para os componentes:

- **contador variável** crescente, recebe uma mensagem "increment" (2);
- **constante numérica**, recebe uma mensagem "action" (4);
- **multiplicador**, que duplica cada mensagem recebida em dois eventos; recebe uma mensagem "action" (6).

O componente contador variável incrementa seu valor em uma unidade e gera um evento "action" com o valor do contador agregado. Este evento dispara o envio de uma mensagem (por meio de um *link*), para modificação de propriedade x do componente **par**

ordenado (3), cujo valor é o do contador (extraído a partir do evento). Deste modo, o contador representará o valor de x que se desloca pelo domínio.

As mensagens “action”, recebidas pelo componente constante numérica, disparam o envio de mensagens para a modificação da propriedade, associada ao primeiro operando (“OperandA”), do componente **soma** (5). Agregado a esta mensagem vai o valor da constante.

Através de um *link*, o componente multiplicador redireciona os eventos (em dobro) na forma de mensagens “increment” para outro contador variável (7). Para cada mensagem “increment” recebida, este segundo contador envia uma mensagem para modificação de propriedade do segundo operando (“OperandB”) do componente soma (8). Por receber o dobro das mensagens provenientes do relógio, este segundo contador sempre terá registrado o dobro da contagem (em relação ao primeiro contador), comportando-se como a parte 2x da função.

Ao receber o valor de seus dois operandos (um proveniente da constante numérica outro do contador dobrado), o componente soma, por sua vez, realiza a soma desses operandos e gera um evento “action”, com o resultado da operação agregado. Esta soma completa o cálculo da função. Como, no exemplo, o contador dobrado representa 2x e a constante possui o valor 3, obtemos 2x+3. Por este motivo, este evento dispara o envio de uma mensagem (por meio de um *link*), para modificação da propriedade y do componente par ordenado (9).

Ao receber os valores de x e y, o componente par ordenado gera um evento “action” que dispara a mensagem “insert point”, acompanhada dos valores de x e y, extraídos do evento, que é enviada para o componente **plano cartesiano** (10). Cada ponto que ele recebe através de mensagem é inserido no gráfico como parte da curva.

Já que é possível configurar o intervalo de tempo do relógio, todo o processo pode ser observado em qualquer velocidade. Cada componente se comporta de forma ativa, apresentando visualmente seu valor de estado a cada mudança. Isto permite ao aluno observar como cada elemento interfere na dinâmica da formação da curva.

Através da folha XSL apresentada no Apêndice C.4, a composição pode ser apresentada em uma página *Web*. Isto propicia a elaboração de um material dinâmico e interativo, por parte dos professores, além de possibilitar aos alunos a apresentação e intercâmbio de produções através da rede.

5.2. Casa Mágica e *Anima*

Os componentes apresentados no tópico 5.1 são uma amostra da biblioteca disponível no sistema. Através deles é possível representar essa mesma, ou outras funções de diversas formas. A construção de representações como essa, por parte do aluno, permite não apenas uma reflexão individual sobre o tema, mas possibilita a socialização de produções construídas por diversos alunos, a fim de promover uma reflexão em grupo.

Na explanação do funcionamento dos modelos que simulam funções, foram detalhados aspectos da sua elaboração, utilizando componentes de *software*. Houve um cuidado em explicitar como cada relação entre os objetos se desenvolve, a fim de verificar precisamente como o modelo *Anima* atua no projeto proposto. Isto pode causar a sensação de que a elaboração de tais composições pode se tornar complexa para o usuário.

A maior parte do que foi apresentado ocorre por trás dos bastidores, sem que o usuário se envolva com os detalhes do funcionamento. O sistema Casa Mágica realiza a ligação entre os componentes de forma visual e simples (isto é apresentado no tópico 2.4). As escolhas de: evento estímulo, mensagem, categoria e tipo são feitas a partir de listas de seleção, que apresentam apenas as opções declaradas na descrição CIM pertinentes ao contexto.

O recurso de múltiplas 'visões' (`views`) permite a seleção e apresentação das propriedades do componente que são relevantes para o contexto. Assim, mesmo que o componente possua uma extensa quantidade de propriedades, que poderiam tornar sua configuração bastante complexa, o usuário verá apenas algumas delas, necessárias para seu trabalho. As 'visões' podem ser combinadas com os 'visores' do sistema Casa Mágica, apresentados no tópico 2.4.1, que permitem a construção de janelas personalizadas para a edição de propriedades.

Além disto, apesar da representação interna de identificação de classes, atributos, eventos, mensagens, etc. ser feita em inglês, para fins de internacionalização, a representação CIM possui opções de tradução para todos eles. Deste modo, o usuário que está construindo a aplicação pode enxergar tudo na sua língua, enquanto que internamente o sistema mapeia para a identificação original.

O sistema Casa Mágica já dispunha de alguns dos recursos utilizados neste projeto, *Anima* porém acrescenta uma série de benefícios:

- Toda a estrutura que envolve CIM (classes, interfaces e metadados), composições e mensagens foi construída utilizando um formato aberto,

independente de plataforma e implementação. Os esquemas apresentados nesta dissertação também são abertos e públicos. Isto não limita o resultado da produção ao ambiente Casa Mágica, mas permite que qualquer outra aplicação tenha acesso ao mesmo. A integração pode ser estática, através do acesso ou modificação dos arquivos *Anima*, ou dinâmica, pela troca de mensagens entre objetos, utilizando mensagens *Anima*.

- Através deste modelo aberto, também é possível que o produto de outros sistemas seja inserido ou editado no sistema Casa Mágica. Tal como uma ferramenta de autoria, o ambiente pode integrar produtos de outros sistemas mais especializados.
- A estrutura das composições, baseada em XML, em conjunto com as folhas XSL, mostrou-se excelente para a sua integração com a *Web*. Deste modo, foi possível combinar XML, XSL e uma linguagem de código móvel (Java), para que as produções possam trafegar livremente na forma de páginas.
- Através da representação CIM, foi enriquecida a descrição das classes disponíveis no sistema Casa Mágica, pois ele dispunha apenas de recursos para codificação de classes equivalentes à da maioria das linguagens de programação, sem mecanismos para definição de metadados, descrições multilíngua, etc.

Através deste estudo de caso é possível observar um aspecto discutido no capítulo anterior. O modelo *Anima* define uma metodologia particular para estabelecer a relação entre os objetos, através de *links* que fazem associações do tipo objeto_origem/evento→objeto_destino/método, sem que seja necessária a inserção de código de programação. Se por um lado, como já foi abordado no tópico 4.2.1, esta modalidade estabelece um tipo restrito de relação entre os objetos, o estudo de caso demonstra que, quando os componentes são devidamente projetados para trabalhar conforme esta filosofia, os alunos e professores podem construir uma composição sem se envolver com detalhes de uma linguagem de programação.

Em um ambiente onde se desenvolve uma atividade de ensino-aprendizagem, a simplicidade no manuseio da ferramenta é estratégica, já que se pretende dar ênfase ao objeto de estudo (no exemplo apresentado, a matemática) e não a detalhes do sistema computacional envolvido no processo.

As limitações encontradas neste projeto – decorrentes do modelo de ligação entre objetos – foram superadas projetando-se componentes que desempenham tarefas, e eventualmente seriam solucionadas com a inserção de código em tempo de produção.

Na segunda composição apresentada no estudo de caso, por exemplo, havia um problema no modo como combinar o resultado do contador, que representa o elemento x da função, com o resultado do somador, que representa o f(x) em um par ordenado, a ser enviado para o componente que representa o plano cartesiano. Isto foi resolvido, projetando-se um componente adicional, cuja função específica é juntar os dois valores em um par ordenado.

Do ponto de vista de uma ferramenta para construção de sistemas, voltada para profissionais de informática, este tipo de estratégia pode não ser adequado, uma vez que significa um *overhead* em termos de memória e/ou processamento, porém, no contexto do sistema Casa Mágica e *Anima*, observamos que as composições não sofrem qualquer deficiência em termos de *performance*. Isto se deve à natureza dos projetos que envolvem produção de composições, desenvolvidos em atividades de ensino-aprendizagem, que são geralmente muito simples e raramente são afetados quando a solução, em termos de programação, não é a mais eficiente.

O estudo de caso apresentado neste capítulo, permitiu uma visão abrangente do modo como *Anima* atua em sistemas envolvidos no desenvolvimento de atividades de ensino-aprendizagem, através de um exemplo concreto. No próximo capítulo, concluímos a dissertação, apresentando uma análise abrangente dos resultados obtidos com *Anima* e as perspectivas de evolução do projeto.

6. CONCLUSÃO

Na medida em que a educação se envolve de forma crescente com as novas tecnologias, cresce também uma relação de dependência com os caminhos pelos quais estas se desenvolvem.

Independente do prisma pelo qual entendemos o papel dos meios tecnológicos dentro do processo de ensino/aprendizagem, não há dúvidas que o acelerado avanço da tecnologia afeta diretamente esse processo, seja porque se apresentam novas ferramentas com potencialidades que não foram antes exploradas, ou porque novas tecnologias abrem diferentes perspectivas que modificam concepções tradicionais, sobre as quais estão pautadas nossas atividades de ensino-aprendizagem (tal como a noção de espaço e tempo), etc..

Por este motivo, é um desafio para a educação no Brasil, não apenas encontrar caminhos no universo das novas tecnologias, que nos são dadas prontas, como também propiciar que nos tornemos atuantes na sua concepção.

Se o emprego das novas tecnologias tem repercutido tão intensamente na sociedade moderna, de tal forma que, como afirma Pierre Babin, “modela progressivamente um outro comportamento intelectual e afetivo” [BAB89], cabe a cada povo e cultura contribuir para que estas novas tecnologias traduzam um modo de ser e pensar coletivo, expressão de múltiplas aspirações.

Deste modo, ao mesmo tempo em que se concebem novas metodologias de ensino-aprendizagem, afinadas com o atual mundo tecnológico, é também importante a atuação no processo de construção das novas tecnologias em si.

Se, por um lado, há uma abundância de *software* educacional, por outro, tem havido uma carência de iniciativas em direção a um trabalho amplamente colaborativo, de modo que os esforços possam ser combinados e os resultados não apenas serem usufruídos por todos mas também serem contextualizados, de acordo com as peculiaridades de cada grupo ou região.

A fragmentação das iniciativas conduz não apenas a uma repetição de esforços para a solução do mesmo problema, como também divide projetos futuros instalados sobre bases diferentes.

Um exemplo disto está na multiplicação de ambientes diferentes e incompatíveis que gerenciam a educação à distância. Do ponto de vista comercial, é compreensível que empresas elaborem produtos concorrentes para disputarem um mercado comum; entretanto, no Brasil, ainda são predominantes as iniciativas de universidades que elaboram sistemas completos para gerenciamento de educação à distância, como produto de pesquisas ou para atender a sua necessidade interna – a exemplo do AulaNet[1] [FUK00], desenvolvido pelo Laboratório de Engenharia de Software da PUC-Rio, do TelEduc[2] [ROC00], desenvolvido pelo Núcleo de Informática Aplicada à Educação (Nied) da Unicamp e do LearnLoop[3], traduzido pelo Núcleo Avançado de Computação Sônica e Multimídia da Universidade Federal de Uberlândia. Em geral, esses sistemas não se têm configurado como produtos comerciais e muitos deles têm sido distribuídos publicamente. No entanto, ao contrário do que se poderia esperar, os projetos são independentes. A maioria deles não aproveita o trabalho desenvolvido em outros projetos e os sistemas não se integram.

Diversos outros projetos são desenvolvidos sobre esses sistemas – a exemplo da "Categorização e Estruturação de Mensagens Textuais em Cursos", para o AulaNet [FUK02] e do "Ambiente de Autoria de Cursos à Distância", para o TelEduc [TES00] – acrescentando-lhes funcionalidades que são produto das mais recentes pesquisas realizadas no Brasil. Cada novo progresso, porém, fica encapsulado dentro do sistema em que foi desenvolvido e dificilmente pode ser aproveitado por usuários de outros produtos.

Fora do contexto específico da informática aplicada à educação, não faltam exemplos de iniciativas bem sucedidas na produção de grandes produtos de forma colaborativa. Alguns sistemas operacionais baseados em UNIX são produtos de grande cooperação, de esforços espalhados pelo mundo, que contribuíram na sua codificação, teste e aperfeiçoamento constantes, resultando em produtos robustos e estáveis.

A questão principal consiste no fato de que os sistemas já devem ser pensados como projetos abertos, preparados para a integração e trabalho colaborativo no seu desenvolvimento.

Seguindo esta lógica, o projeto *Anima* adquiriu duas dimensões neste trabalho:

---

[1] A página do AulaNet está disponível no endereço http://anauel.cead.puc-rio.br/aulanet2/ e a página Guia AulaNet, com informações completas sobre o *software* no endereço http://guiaaulanet.eduweb.com.br/.

[2] A página do TelEduc está disponível no endereço http://hera.nied.unicamp.br/teleduc/.

[3] O LearnLoop foi desenvolvido no *The Viktoria Institute* e The Council For IT e usado na Gothenburg Business School em Gothenburg, Suécia e criado por Daniel Önnerby, Per Åsberg e Britt Klintenberg. A página original do projeto está disponível no endereço http://www.learnloop.org/ e a página da versão nacional, no endereço http://www.ead.ufu.br/learnloop/.

Uma primeira dimensão mais imediata, que consistiu em definir um modelo e sua respectiva implementação, que permita explorar os objetos e componentes de *software* em atividades de ensino-aprendizagem.

E uma segunda dimensão cuja importância se apresenta de forma latente – a de definir um modelo aberto, projetado para uma ampla integração e implementado em um formato apto a um desenvolvimento colaborativo.

Do ponto de vista mais imediato, observamos que a implementação de *Anima* dentro do ambiente Casa Mágica, não representou uma simples adaptação de um sistema, mas exigiu que todo o modelo subjacente fosse repensado.

Os objetos e classes cuja semântica se limitava quase que exclusivamente àquela disponível em linguagens e sistemas orientados a objetos, destinados a profissionais de informática, receberam uma nova roupagem. O enfoque se deslocou e, ao invés de esses serem tomados como peças acessórias, passaram a ser o centro do projeto. Conseqüentemente, adquiriram uma semântica adicional, onde se acrescentaram aspectos como:

Descrição das características e funcionalidades dos recursos para usuários

Como acontece na maioria dos sistemas, os objetos e classes só possuem agregados a eles informações suficientes para sua compreensão por um especialista em programação de computadores e para sua interpretação pelo computador.

No caso específico do sistema Casa Mágica, qualquer informação referente aos recursos (objetos, classes, locais, etc.) destinada aos usuários, era representada por vias indiretas e não estava agregada aos recursos em si.

Anima estabeleceu um mecanismo de associação direta das informações destinadas aos usuários, com os recursos, utilizando técnicas de agregação de metadados. Hoje, descrições sobre: componentes, classes, composições, etc. estão amplamente disponíveis aos usuários, com a opção de escreverem-se em múltiplas línguas.

A associação direta dos metadados com os recursos permite que aqueles sejam facilmente acessados, e de modo uniforme, por qualquer módulo do sistema. Além do que, isto viabiliza a centralização das ferramentas responsáveis pela criação e edição desses metadados. Estas ferramentas só precisam ser capazes de manipular o formato aberto e padronizado em que eles são representados.

Descrição dos objetos do ponto de vista educacional

O sistema Casa Mágica ainda não dispunha de qualquer mecanismo para a descrição de seus objetos, do ponto de vista educacional. *Anima* não apenas estabeleceu o modo de fazê-lo, como adotou um modelo de descrição baseado em padrões internacionais (IMS e LOM) e integrado com os demais metadados do objeto.

Na atual versão, o sistema Casa Mágica aproveita apenas parte de todo o potencial dos metadados educacionais, oferecendo ao usuário a possibilidade de consultar informações referentes à aplicação pedagógica dos componentes disponíveis no ambiente, as quais são extraídas desses metadados.

Em versões futuras do sistema, planejamos a implementação de uma infra-estrutura que permita a seleção e classificação baseada em metadados educacionais. Isto pode ser explorado localmente – na estação do usuário – e no ambiente *Web*, dado que a representação em XML ou RDF é adequada para ambos.

Além da sua relevância para os ambientes de produção, como o sistema Casa Mágica, os metadados educacionais também serão importantes para bibliotecas que disponibilizam componentes na *Web*, constituindo um subsídio uniforme para documentação, classificação e seleção dos componentes.

Outros benefícios imediatos, observados na inserção do modelo *Anima* dentro do sistema Casa Mágica, se mesclam com a dimensão latente abordada anteriormente. Em torno de cada elemento do modelo *Anima* (composições, CIM, mecanismo de transformação e protocolo de mensagens) giram benefícios imediatos e latentes.

Representação da Composição

No capítulo 2, ressaltamos a crescente necessidade de integração entre aplicações educacionais. A representação da composição foi projetada para se comportar como um espaço central e público, onde as aplicações podem trabalhar coletivamente em torno do mesmo produto.

Ferramentas especializadas, como o Modellus, Winplot e Homos, poderão produzir resultados na forma de composições *Anima*. O produto resultante pode ser de diversas granularidades, conforme a conveniência do *software*:

- A granularidade maior seria um único e grande componente, que contém todo o produto e é capaz de se integrar por interfaces *Anima* com outros componentes. Sua estrutura interna não é acessível.
- A granularidade menor seria uma coleção de pequenos objetos/componentes interligados por *links*, conforme o modelo de composições *Anima*. Cada um dos objetos/componentes que fazem parte do produto se comunica por interfaces *Anima*.

Ferramentas mais genéricas, como o sistema Casa Mágica, serão capazes de combinar os resultados de outras aplicações ou, quando as composições são produzidas em níveis de granularidade menor, modificar seu conteúdo.

Por este motivo, se por um lado produzir composições de granularidade menor pode ser, a princípio, mais difícil para aplicações especializadas adaptadas para *Anima*, por outro, o resultado permitirá uma interferência maior do usuário.

Representação de Classe, Interface e Metadados (CIM)

Do mesmo modo que a representação da composição, no seu domínio de atuação CIM se configura como um foro aberto para o compartilhamento de classes.

Combinadas, as classes produzidas em linguagens de código móvel e representações CIM resultam em pacotes altamente autônomos e portáveis, não apenas no que diz respeito a sua execução, como também em relação às informações necessárias para sua compreensão e seu uso.

Do ponto de vista da integração entre aplicações, no intercâmbio de composições, os objetos que as compõem poderão vir acompanhados de documentação, através da representação CIM da classe. Isto permite o entendimento das peças de uma composição independente do ambiente que as produziu.

Na medida em que uma ferramenta adapta sua estrutura para a produção de composições *Anima*, terá que transformar grande parte de seu código em classes, que serão publicamente distribuídas e funcionarão de forma autônoma em relação a seu sistema de origem, uma vez que estas classes acompanharão as composições.

Indiretamente, este fenômeno pode motivar a reunião de esforços para solução de problemas comuns. Deste modo, duas aplicações que fazem uso de gráficos traçados no plano cartesiano, podem compartilhar o mesmo componente que realiza esta tarefa.

Mecanismos de Conversão

Os mecanismos de conversão – independentes de plataforma e implementação – são a chave para tornar as composições efetivamente autônomas. Eles deslocam a 'inteligência' do processo de execução das composições para uma esfera neutra, independente da aplicação que as criou. Como já foi abordada, esta autonomia é essencial para integração de diferentes sistemas.

Para uma mesma composição é possível produzirem-se folhas de transformação que resultem em apresentações diferentes. Deste modo, uma composição baseada em componentes Java pode, através de uma folha de estilo, dar origem a uma aplicação Java independente ou, através de outra folha, resultar em uma página *Web* que apresenta a composição. Isto será feito de forma independente da ferramenta (ou ferramentas) que produziu a composição.

Protocolo Padronizado de Comunicação

No estudo de caso apresentado no capítulo 5, apresentamos uma aplicação que permite a exploração visual de funções matemáticas. Os componentes são construídos em Java e podem ser combinados dentro do ambiente Casa Mágica.

No capítulo 4, destacamos que um dos fatores que motivaram a elaboração de *Anima*, foi a possibilidade de integração entre objetos de diferentes linguagens e implementações. De fato, o projeto do capítulo 5 pode ser enriquecido, por exemplo, através da sua integração com componentes produzidos no sistema Alice, discutido no capítulo 2.

Alice é um sistema que permite a construção de cenários tridimensionais. Ele é construído sobre a linguagem de programação Python [ROS01], que é orientada a objetos e possui todos os subsídios necessários para a implementação de *Anima*. Deste modo, Alice aceita a inserção direta de comandos Python em seu código.

Utilizando o protocolo padronizado de comunicação é possível, através de um canal TCP/IP, integrar uma composição de objetos Java com outra de objetos Python/Alice. Como ambos oferecem suporte à execução em uma página *Web*, esta pode ser o ambiente de integração. Assim, podemos construir uma máquina capaz de controlar o movimento de um corpo que se desloca no ambiente tridimensional de Alice. As composições Java e Alice são executadas em espaços de trabalho independentes e, através da troca de mensagens *Anima*, interagem entre si.

A integração das duas produções pode ser feita no ambiente Casa Mágica. Como ele não é capaz de instanciar objetos escritos em Alice, o sistema apresenta, em tempo de produção, apenas uma fotografia do objeto. A partir das informações disponíveis na representação CIM, a ferramenta possui elementos suficientes para a integração do objeto Alice, mesmo que ele não possa ser instanciado.

Através deste exemplo, procuramos mostrar a importância da existência de um protocolo padronizado de comunicação quando desejamos integrar objetos de diferentes linguagens e implementações.

A estrutura apresentada neste trabalho fornece subsídios ao desenvolvimento de ferramentas, para que muitas outras linguagens e sistemas possam se conectar a *Anima*, além de Java e da linguagem desenvolvida no sistema Casa Mágica (já implementados). Identificamos plataformas e linguagens preferenciais para sua expansão, tais como: Alice, Squeak, VRML, Javascript e C++.

Para linguagens que não são orientadas a objetos, como VRML, pode-se conceber uma camada abstrata de classes através da representação CIM, que são posteriormente mapeadas para a representação convencional VRML por folhas XSLT. Recursos de programação avançados não disponíveis nesta linguagem serão complementados através da sua integração com outra linguagem, como Java.

Durante todo o desenvolvimento deste trabalho, pudemos antever diversas demandas que darão origem a projetos futuros.

Além das que já foram abordadas neste capítulo, observamos a necessidade de um detalhamento mais profundo da caracterização de grupos de mensagens compatíveis entre emissor e receptor, estabelecendo um mecanismo mais robusto de conversão e adaptação. Planejamos utilizar RDF para definir uma estrutura que explore a hierarquia de classes, na caracterização de famílias de mensagens. Esta hierarquia servirá como guia para definir compatibilidade e estratégias de conversão entre mensagens, além de agregar a elas uma semântica adicional àquela proporcionada pela simples descrição de seu rótulo e seus parâmetros.

Retomando a discussão referente à integração de ambientes que gerenciam educação à distância, *Anima* pode ser utilizado como modelo para a produção de sistemas deste tipo, mais abertos e aptos a uma construção participativa.

Ao tratar do futuro da educação on-line, Stephen Downes afirma:

> O modelo predominante para o projeto de cursos irá se assemelhar à arquitetura dos computadores modernos. Haverá um *backbone* (espinha dorsal), análogo à placa mãe do computador, que estabelece a estrutura básica do curso. No *backbone* serão conectados vários módulos de estudo, ferramentas de comunicação, e sistemas de informações do estudante. [DOW98]

Segundo Downes, ao contrário da estrutura predominante dos sistemas para cursos on-line, no futuro haverá fabricantes especializados na construção de módulos com tarefas específicas, tais como e-mail, *chat*, apresentação de simulações, etc. Estes módulos se integrarão através da troca de 'objetos educacionais', conforme padrão LOM – *Learning Object Metadata* e sua respectiva versão XML, definida pelo IMS.

Deste modo nos beneficiaremos com a combinação das melhores soluções em cada área e o aluno terá a liberdade de escolher a opção de módulo que mais lhe aprouver, para o desenvolvimento de determinada tarefa.

A integração de componentes de porte médio e grande, em um sistema para gerenciar cursos on-line, exige uma extensão do modelo definido em *Anima*. Neste caso, será necessária especial atenção aos objetos de dados que irão circular entre os componentes. Muitas destas mensagens conterão estruturas altamente especializadas, tais como: mensagens de correio eletrônico e fórum, respostas dadas por um aluno a uma avaliação, informações cadastrais de um aluno, etc..

Por este motivo, além da estrutura de integração proporcionada por Anima, faz-se necessária a definição de um padrão para a representação dos objetos de informação que serão registrados e transmitidos, tal como está descrito em [SAN99.2].

Pretendemos ainda, em futuras versões, permitir que objetos de diferentes linguagens compartilhem a mesma área de trabalho. Isto implicaria a criação de um *container* gráfico para padronizar o mecanismo de apresentação visual dos objetos e a interação do usuário com o mesmo, sem estabelecer limitações de áreas de apresentação e com possibilidade de sobreposição visual dos objetos. Adicionalmente, seria necessária a criação de um mediador especializado no gerenciamento da interação, do ponto de vista visual que os objetos gráficos teriam. Por exemplo, o controle de colisão e sobreposição de objetos.

Apesar do projeto *Anima* possuir um modelo que abrange o ciclo completo, partindo do planejamento, desenvolvimento, até à execução de composição, esteve claro, desde o início do projeto, que, ao completarmos sua primeira versão, não estaríamos estabelecendo

uma solução definitiva. Pelo contrário, sua estrutura absolutamente aberta, pautada em padrões da *Web* – que além de abertos são extensíveis – viabiliza sua expansão e refinamento.

Esta concepção de um projeto planejado para ser aberto e participativo traduz a crença que defendemos no início deste capítulo, de que nestes caminhos convergentes entre informática e educação, em que se entrelaçam relações complexas e interdependências, atuar de forma colaborativa e contextualizada deixou de ser uma opção e passou a ser um compromisso, necessário para que nossos esforços possam efetivamente contribuir para um futuro melhor na educação.

7. REFERÊNCIAS BIBLIOGRÁFICAS

[ADL01] Adler, Sharon et al. **Extensible Stylesheet Language (XSL) Version 1.0 – W3C Recommendation 15 October 2001**. W3C - World Wide Web Consortium, 2001 [online] http://www.w3.org/TR/xsl/. Arquivo capturado em 3/02/2002.

[ALV95] Alvestrand, Harald Tveit. **RFC1766 – Tags for the Identification of Languages**. UNINETT, 1995 [online] http://www.ietf.org/rfc/rfc1766.txt.Arquivo capturado em 12/02/2002.

[AND00] Anderson, Thor; Wason, Tom. **IMS Learning Resource Meta-data Information Model - Version 1.1 - Final Specification**. IMS Global Learning Consortium, Inc., June 5, 2000 [online] http://www.imsproject.com/metadata/mdinfov1p1.html. Arquivo capturado em 07/09/2000.

[AND01] Anderson, Thor; McKell, Mark. **IMS Learning Resource Meta-data - Version 1.2 Final Specification**. IMS Global Learning Consortium, Inc., May 17, 2001 [online] http://www.imsproject.org/metadata/imsmdv1p2/imsmd_infov1p2.html. Arquivo capturado em 18/09/2001.

[ANP00] Andrews, Peter et al. **Quick: XML Made Easy – Release 3.3.1**. October 3, 2000 [online] http://jxquick.sourceforge.net/quick3/. Arquivo capturado em 28/01/2002.

[BAB89] Babin, Pierre & Kouloumdjian, Marie-France. **Os novos modos de compreender**. São Paulo: Edições Paulinas, 1989.

[BAS] Bastiaan, Martin Koning. **Connected and Scalable: A revolutionary Structure for Online Communities**. EOE – Educational Object Economy.

[BEC01] Beckett, Dave & Miller, Eric & Brickley, Dan. **Expressing Simple Dublin Core in RDF/XML**. Dublin Core Metadata Initiative, November 28, 2001 [online] http://www.dublincore.org/documents/2001/11/28/dcmes-xml/. Arquivo capturado em 10/03/2002.

[BER98] Berners-Lee, Tim et al. **Uniform Resource Identifiers (URI): Generic Syntax**. The Internet Society, 1998 [online] http://www.ietf.org/rfc/rfc2396.txt. Arquivo capturado em 13/02/2002.

[BRA99] Bray, Tim & Hollander, Dave & Layman, Andrew (editor). **Namespaces in XML – World Wide Web Consortium Note 14-January-1999**, W3C - World Wide Web Consortium, January 14, 1999 [online] http://www.w3.org/TR/1999/REC-xml-names-19990114/. Arquivo capturado em 04/03/2002.

[BRA00] Bray, Tim et al. (editor). **Extensible Markup Language (XML) 1.0 (Second Edition) - W3C Recommendation 6 October 2000**. W3C - World Wide Web

Consortium, October 6, 2000 [online] http://www.w3.org/TR/2000/REC-xml-20001006. Arquivo capturado em 15/03/2002.

[BRI00] Brickley, Dan; Guha, R.V. **Resource Description Framework (RDF) Schema Specification 1.0 - W3C Candidate Recommendation 27 March 2000**. World Wide Web Consortium (W3C), March 27, 2000 [online] http://www.w3.org/TR/2000/CR-rdf-schema-20000327/. Arquivo capturado em 20/09/2001.

[CHI97] ChiMu Corporation. **MONDO: A General Architecture for Encoding, Modeling, and Processing Object-Oriented Information Models**. ChiMu Corporation, 1997, [online] http://www.chimu.com/projects/mondo/design/mondoDesign.pdf. Arquivo capturado em 29/01/2002.

[COL99] Colan, Mark. **InfoBus 1.2 Specification**. Sun Microsystems, 1999 [online] http://java.sun.com/products/javabeans/infobus/spec12/IB12Spec.htm. Arquivo capturado em 19/02/2002.

[DIS97] diSessa, Andrea A. **20 Reasons Why You Should Use BOXER (instead of Logo)**. Sixth European Logo Conference, August 20-23, 1997 [online] ftp://soe.berkeley.edu/pub/boxer/Distribution/20reasons.pdf. Arquivo capturado em 15/03/2002.

[DIS01] diSessa, Andrea A. **The Web/comp project – Proposal**. School of Education, University of California, Berkeley, 2001 [online] http://www.soe.berkeley.edu/~boxer/webcomp/proposal.html. Arquivo capturado em 8/12/2001.

[DOD01] Dodds, Leigh. **The Collected Works of SAX**. XML.com, July 18, 2001 [online] http://www.xml.com/pub/a/2001/07/18/collectedsax.html. Arquivo capturado em 2/02/2002.

[DOW98] Downes, Stephen. **The Future of Online Learning**. Online Journal of Distance Learning Administration, Volume I, Number 3, Fall 1998, State University of West Georgia, Distance Education Center [online] http://www.westga.edu/~distance/downes13.html. Arquivo capturado em 15/06/2002.

[FAL01] Fallside, David C. (editor). **XML Schema Part 0: Primer – W3C Recommendation, 2 May 2001**. W3C - World Wide Web Consortium, 2001 [online] http://www.w3.org/TR/xmlschema-0/. Arquivo capturado em 10/03/2002.

[FER89] Ferreira, Aurélio B. H. **Minidicionário da Língua Portuguesa**. Rio de Janeiro, Editora Nova Fronteira, 1989.

[FRE96] Freed, Ned & Borenstein, Nathaniel S. **RFC2046 – Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types**. Innosoft International /

First Virtual Holdings, 1996 [online] http://www.ietf.org/rfc/rfc2046.txt. Arquivo capturado em 9/02/2002.

[FUG98] Fuggeta, Alfonso & Picco, Gian Pietro & Vigna, Giovanni. **Understanding Code Mobility**. IEEE Transactions on Software Engineering, volume 24, 1998 [online] http://www.elet.polimi.it/Users/DEI/Sections/Compeng/GianPietro.Picco/papers/tse98.ps.gz. Arquivo capturado em 15/03/2002.

[FUK00] Fuks, Hugo. **Aprendizagem e Trabalho Cooperativo no Ambiente AulaNet**. Rio de Janeiro: Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro, Fevereiro de 2000 [online] http://139.82.193.20/aulanet2/misc/aprendizagem.pdf. Arquivo capturado em 15/03/2002.

[FUK02] Fuks, Hugo & Gerosa, Marco Aurélio & Lucena, Carlos José Pereira de. **Usando a Categorização e Estruturação de Mensagens Textuais em Cursos pelo Ambiente AulaNet**. Revista Brasileira de Informática na Educação – Sociedade Brasileira de Computação, número 10, Abril de 2002 [online] http://139.82.193.20/aulanet2/misc/CategSBCIE.pdf. Arquivo capturado em 15/03/2002.

[GEL99] Gellersen, Hans-W. Gaedke, Martin. **Object-Oriented Web Application Development.** IEEE Internet Computing, volume 3, number 1, p. 60-68, January-February 1999.

[GRE98] Green, Dale. **The Reflection API** in Campione, Mary. The Java Tutorial Continued: The Rest of the JDK. Addison-Wesley, 1998 [online] http://java.sun.com/docs/books/tutorial/reflect/index.html. Arquivo capturado em 30/01/2002.

[HAM97] Hamilton, Graham (editor). **JavaBeans 1.01 specification**. Sun Microsystems, 1997 [online] http://java.sun.com/products/javabeans/docs/beans.101.pdf. Arquivo capturado em 8/02/2002.

[HOD02] Hodgins, Wayne (chair). **Draft Standard for Learning Object Metadata**. IEEE Learning Technology Standards Committee (LTSC), January 18, 2002 [online] http://ltsc.ieee.org/doc/wg12/LOM_WD6_3a.pdf. Arquivo capturado em 7/02/2002.

[HOL92] Holmevik, Jan Rune. **Compiling SIMULA: a historical study of technological genesis**. IEEE Annals of the History of Computing, 16 (4), p. 25-37, 1994 [online] http://www.cis.um.edu.mt/~jskl/simula.html. Arquivo capturado em 15/03/2002.

[HUG97] Hugunin, Jim. **Python and Java: The Best of Both Worlds**. Proceedings of the 6th International Python Conference, October 14, 1997 [online] http://www.python.org/workshops/1997-10/proceedings/index.html. Arquivo capturado em 12/02/2002.

[ING81] Ingalls, Daniel H. H. **Design Principles Behind Smalltalk**. BYTE Magazine, August 1981.

[ISO86] ISO – International Organization for Standardization**. ISO 8879:1986(E), Information processing - Text and Office Systems - Standard Generalized Markup Language (SGML)**, First edition - 1986-10-15. Geneva: International Organization for Standardization, 1986.

[ISO96] ISO – International Organization for Standardization, International Electrotechnical Commission. **ISO/IEC 10179:1996. Document Style Semantics and Specification Language (DSSSL)**. International Organization for Standardization, International Electrotechnical Commission, 1996.

[JOH] Johnson, Jeff. **Object Serialization, Persistence, and Distribution: An Overview**. Brigham Young University – Hawaii Campus [online] http://www.cs.byuh.edu/~johnsonj/research/serialization/serial_paper.html. Arquivo capturado em 12/06/2002.

[JOH98] Johnson, Ray. **Tcl and Java Integration**. Sun Microsystems Laboratories, 1998. [online] http://tcl.activestate.com/software/java/tcljava.pdf. Arquivo capturado em 12/02/2002.

[JOH99.1] Johnson, Mark. **XML JavaBeans**, JavaWorld, volume 4, issue 2-7, February-July 1999 [online] http://www.javaworld.com/javaworld/jw-02-1999/jw-02-beans.html. Arquivo capturado em 29/01/2002.

[JOH99.2] Johnson, Mark. **Bean Markup Language**, JavaWorld, volume 4, issue 8-10, August-October 1999 [online] http://www.javaworld.com/javaworld/jw-08-1999/jw-08-beans.html. Arquivo capturado em 29/01/2002.

[KRE98] Kreutzer, Wolfgang. **Basic Aspects of Squeak and the Smalltalk-80 Programing Language.** July 1998 [online] http://kaka.cosc.canterbury.ac.nz/~wolfgang/cosc205/smalltalk1.html. Arquivo capturado em 15/03/2002.

[LAK80] Lakoff, George & Johnson, Mark. **Metaphors We Live By**. Chicago: University of Chicago Press, 1980.

[LAS99] Lassila, Ora; Swick, Ralph R. **Resource Description Framework (RDF) Model and Syntax Specification - W3C Recommendation 22 February 1999**, W3C - World Wide Web Consortium (W3C), February 22, 1999 [online] http://www.w3.org/TR/REC-rdf-syntax/. Arquivo capturado em 20/09/2001.

[LTS00] IEEE Learning Technology Standards Committee (LTSC). **Draft Standard for Learning Object Metadata**. New York: Institute of Electrical and Electronics Engineers, Inc., February 6, 2000 [online] http://ltsc.ieee.org/doc/wg12/LOM_WD4.PDF. Arquivo capturado em 07/09/2000.

[MAC01] Machado, Nilson José. **Matemática e Educação – Alegorias, tecnologias e temas afins**. São Paulo, Cortez, 2001.

[MAN97] Manola, Frank (editor). **NICTS Technical Committee H7 Object Model Features Matrix**. National Committee for Information Technology Standards (NCITS), X3H7-93-007v12b, May 25, 1997 [online] http://www.objs.com/x3h7/h7home.htm. Arquivo capturado em 15/03/2002.

[MAN98] Manola, Frank. **Towards a Web Object Model**. Object Services Consulting, Inc. (OBJS), 1998 [online] http://www.objs.com/OSA/wom.htm. Arquivo capturado em 15/03/2002.

[MAN98.2] Manola, Frank. **Some Web Object Model Construction Technologies**. Object Services Consulting, Inc. (OBJS), 1998 [online] http://www.objs.com/OSA/wom-II.htm. Arquivo capturado em 15/03/2002.

[MAN99] Manola, Frank. **Technologies for a Web Object Model**. IEEE Internet Computing, volume 3, number 1, p. 38-47, January-February 1999 [online] http://www.objs.com/survey/wom-ieee.htm. Arquivo capturado em 15/03/2002.

[MIL99] Milne, Philip & Walrath, Kathy. **Long-Term Persistence for JavaBeans**. Sun Microsystems, 1999 [online] http://java.sun.com/products/jfc/tsc/articles/persistence/. Arquivo capturado em 30/01/2002.

[MOZ00] The Mozilla Organization. **Rhino Overview**. The Mozilla Organization, 2000 [online] http://www.mozilla.org/rhino/overview.html. Arquivo capturado em 12/02/2002.

[PAP85] Papert, Seymour. **LOGO: Computadores e Educação**. São Paulo: Brasiliense, 1985.

[PAP94] Papert, Seymour. **A Máquina das Crianças – Repensando a Escola na Era da Informática**. Porto Alegre: Artes Médicas, 1994.

[RAG97] Raggett, Dave. **HTML 3.2 Reference Specification - W3C Recommendation 14-Jan-1997**, W3C - World Wide Web Consortium, January 14, 1997 [online] http://www.w3.org/TR/REC-html32.html. Arquivo capturado em 15/03/2002.

[REP96] Repenning, Alexander & Ambach, James. **Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing**, Proceedings of the 1996 IEEE Symposium of Visual Languages, Boulder, CO, Computer Society, pp. 102-109, 1996 [online] http://www.cs.colorado.edu/~l3d/systems/agentsheets/Documentation/VL-96-Paper.pdf. Arquivo capturado em 15/03/2002.

[REP99] Repenning, Alexander & Ioannidou , Andri & Phillips, Jonathan. **Collaborative Use & Design of Interactive Simulations**. Proceedings of Computer Supported Collaborative Learning Conference at Stanford (CSCL'99), 1999 [online]

http://www.cs.colorado.edu/%7Eralex/papers/PDF/CSCL99.pdf. Arquivo capturado em 15/03/2002.

[RID99] Rideau, François-René. **Metaprogramming and Free Availability of Sources – Two Challenges for Computing Today**. Bretagne: January 25-27, 1999 [online] http://fare.tunes.org/articles/ll99/mpfas.html. Arquivo capturado em 12/02/2002.

[ROC00] Rocha, Heloisa Vieira da; Romani, Luciana Alvim Santos; Silva, Celmar Guimarães da. **O Ambiente para Ensino a Distância TelEduc**. Palestra apresentada no Centro de Computação (CCUEC), Unicamp, Dezembro de 2000 [online] http://hera.nied.unicamp.br/teleduc/palestras/ccuec_dez_2000_heloisa/index.htm . Arquivo capturado em 15/03/2002.

[ROS98] Roschelle, J. Digiano, C. Pea, R. Kaput, J. **Educational Software Components of Tomorrow (ESCOT)**. SRI International, 1998 [online] http://www.escot.org/docs/MSET_ESCOT.html. Arquivo capturado em 12/02/2002.

[ROS98.2] Roschelle, Jeremy & Kaput, Jim & Stroup, W. **SimCalc: Accelerating students' engagement with the mathematics of change**. In M. Jacobson & R. Kozma (Eds.), Educational technology and mathematics and science for the 21st century. Hillsdale, NJ: Erlbaum, 1998 [online] http://www.simcalc.umassd.edu/NewWebsite/downloads/MathofChange.pdf. Arquivo capturado em 15/03/2002

[ROS99] Roschelle, Jeremy et al. **Developing Educational Software Components**. IEEE Computer Society – Computer, volume 32, number 9, September, 1999.

[ROS01] Rossum, Guido van. **Python Tutorial**. PythonLabs, 21 de Dezembro de 2001 [online] http://www.python.org/doc/current/tut/tut.html. Arquivo capturado em 16/03/2002.

[SAN98] Santanchè, André. **Sistema para Construção de Aplicações Educacionais**. IV Congresso Ibero-Americano de Informática na Educação, 1998 [online] http://www.geocities.com/santanche/publicado/ArtigoRIBIE1998.pdf. Arquivo capturado em 7/02/2002.

[SAN99] Santanchè, André & Teixeira, Cesar Augusto Camillo. **Integrando Instrucionismo e Construcionismo em Aplicações Educacionais através do Casa Mágica**. V Workshop de Informática na Escola – XIX Congresso da SBC, 20 de Julho de 1999 [online] http://www.geocities.com/santanche/publicado/WIE99.pdf. Arquivo capturado em 15/03/2002.

[SAN99.2] Santanchè, André & Teixeira, Cesar Augusto Camillo. **Explorando Linguagens de *Markup* Extensíveis na Construção de Sistemas de Educação Baseados na *Web***. XXVI Seminário Integrado de Software e Hardware – XIX Congresso da SBC, 20 de Julho de 1999 [online]

http://www.geocities.com/santanche/publicado/SEMISH1999.pdf. Arquivo capturado em 15/06/2002.

[SAN00] Santanchè, André & Teixeira, Cesar Augusto Camillo. **Construindo e explorando o conhecimento através de Componentes Educacionais embutidos em hiperdocumentos**. VI Workshop de Informática na Escola, 17 de Julho de 2000 [online] http://www.geocities.com/santanche/publicado/WIE2000.pdf. Arquivo capturado em 7/02/2002.

[SIL98] Silberzahn, Nicolas. **Dealing with the Electronic Patient Record variability: Object Oriented XML**. Workshop "SGML/XML in Healthcare" - GCA SGML/XML Europe ´98 Conference Paris, May 18, 1998 [online] http://www.digitalairways.com/NiS/ParisXML98/. Arquivo capturado em 15/03/2002.

[STE96] Steindl, Christoph. **Reflection in Oberon**. 6th ECOOP Workshop for PhD Students in Object-Oriented Programming, Linz, Austria, July 7-8, 1996.

[SUN01] Sun Microsystems. **The Java Architecture for XML Binding – User's Guide**. Sun Microsystems, 2001 [online] http://java.sun.com/xml/jaxb/jaxb-docs.pdf. Arquivo capturado em 02/02/2002.

[SZP99] Szperski, Clemens. **Components and Objects Together**. Software Development Magazine, 1999 [online] http://www.sdmagazine.com/documents/s=758/sdm9905b/. Arquivo capturado em 10/02/2002.

[TEO00] Teodoro, Vitor Duarte. **Modellus – Interactive Modelling with Mathematics – version 2.01**. Faculty of Sciences and Technology – New University of Lisbon, 2000.

[TRA96] Travers, Michael D. **Programming with Agents: New metaphors for thinking about computation**. Massachusetts Institute of Technology, June 1996 [online] http://xenia.media.mit.edu/~mt/diss/prog-w-agents.pdf. Arquivo capturado em 15/03/2002.

[TES00] Tessarollo, Marcia Renata Matero. **AutorWeb - Ambiente de Autoria de Cursos à Distância**. Dissertação de Mestrado. Unicamp – Universidade Estadual de Campinas, 2000 [online] http://hera.nied.unicamp.br/teleduc/publicacoes/marcia_disser.pdf. Arquivo capturado em 15/03/2002.

[VAL93] Valente, José Armando. Diferentes Usos do Computador na Educação. In: Valente, José Armando (org.). **Computadores e Conhecimento: repensando a educação**. Campinas: Gráfica da UNICAMP, 1993 [online] http://www.nied.unicamp.br/publicacoes/separatas/Sep1.pdf. Arquivo capturado em 02/11/2001.

[VIN97] Vinoski, Steve. **CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments**. IEEE Communications Magazine, volume 35, number 2, February 1997 [online] http://www.infosys.tuwien.ac.at/Research/Corba/archive/intro/Vinoski-IEEE-CM.pdf. Arquivo capturado em 19/02/2002.

[WEI95] Weibel, Stuart et al. **OCLC/NCSA Metadata Workshop Report**. OCLC/NCSA Metadata Workshop, March 1, 1995 [online] http://dublincore.org/workshops/dc1/report.shtml. Arquivo capturado em 07/06/2002.

[WIL01] Williams, Stuart & Jones, Mark (editors). **XML Protocol Abstract Model – W3C Working Draft 9 July 2001**. W3C - World Wide Web Consortium, 2001 [online] http://www.w3.org/TR/xmlp-am/. Arquivo capturado em 19/02/2002.

[WOO98] Wood, Lauren (WG chair). **Document Object Model (DOM) Level 1 Specification - Version 1.0 - W3C Recommendation 1 October, 1998**, W3C - World Wide Web Consortium, October 1, 1998 [online] http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/. Arquivo capturado em 15/03/2002.

APÊNDICES

A. TÓPICOS DE REVISÃO BIBLIOGRÁFICA

Apresentamos aqui uma síntese de tópicos referentes a tecnologias para *Web*, tratados detalhadamente no relatório técnico "Tecnologias para *Web*", disponível integralmente no endereço:

http://www.nuperc.unifacs.br/santanche/reltecnicos/tecweb.pdf

A.1. XML

XML é uma linguagem de marcação que possui raízes em SGML (*Standard Generalized Markup Language*), ela tem a capacidade de, através do uso de marcadores, agregar informações adicionais a documentos com os mais diversos propósitos. Neste caso, os marcadores são caracterizados pelos símbolos "<" e ">". Deste modo a expressão

```
<indivíduo> <nome>Quincas</nome> tem <idade>15</idade> anos. </indivíduo>
```

permite diferenciar o conteúdo da frase "Quincas tem 15 anos." das marcações (<indivíduo>, <nome> e <idade>), que agregam informações adicionais à frase. Estas informações são facilmente extraídas e isoladas por programas de computador.

Além de linguagem, XML comporta-se também como metalinguagem. Isto significa que possui recursos, por exemplo, para que uma comunidade defina seus próprios marcadores. Através deles, podem definir ou estender linguagens, tornando-as mais apropriadas à representação dos documentos que manipulam.

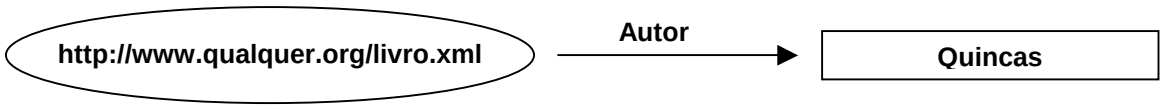
Adicionalmente, pode ser definido um esquema que será seguido na elaboração do documento XML, isto é, feito através de uma DTD – *Document Type Definition*.

A.2. RDF

RDF – *Resource Description Framework* – é uma *framework* baseada em XML, destinada à codificação, troca e reutilização de metadados estruturados.

A idéia básica de RDF [LAS99] não é definir um conjunto universal de metadados e sim prover os mecanismos necessários para que as diversas comunidades codifiquem, troquem e reutilizem metadados estruturados.

Como pode ser observado no diagrama abaixo, RDF utiliza, para descrever um recurso, uma instrução composta de três partes: identificação do recurso, tipo de propriedade e valor da propriedade.



Neste caso, o Autor (tipo de propriedade) do livro definido no documento http://www.qualquer.org/livro.xml (recurso) é Quincas (valor).

A.3. XSLT

O mecanismo de funcionamento da linguagem XSL – *eXtensible Stylesheet Language* [ADL00] se baseia na concepção de um documento XML como uma árvore. XSL cumpre dois papéis: transformar uma árvore de um documento XML em outra árvore (XSLT), e definir uma sintaxe para a árvore resultante, apropriada para a definição da formatação e apresentação do documento (XSL *Formatting Object*).

Para orientar a transformação, um documento XSLT estabelece a forma como subárvores do documento XML de origem devem ser transformadas em subárvores do documento resultante; processadores XSL são capazes de realizar a conversão. É comum a transformação de um documento XML (ou documento escrito em linguagem descendente da metalinguagem XML) em um documento HTML.

B. ESQUEMAS

Apresentamos aqui uma síntese dos esquemas utilizados na definição do modelo abstrato, tratados detalhadamente no relatório técnico "Esquemas do Modelo *Anima*", disponível integralmente no endereço:

http://www.nuperc.unifacs.br/santanche/reltecnicos/esqanima.pdf

Cada esquema é montado em uma tabela, que relaciona nas linhas os elementos que a compõem e suas características. Eventualmente, um elemento pode ser formado de outros elementos. Neste caso, seu tipo será associado a um outro esquema (sub-esquema).

	Esquemas	Sub-esquemas	Elementos
B.1	**Workspace Schema**	Workspace Schema	12
		Object Schema	4
		State Schema	2
		Link Schema	4
B.2	**CIM Schema**	CIM Schema	19
		Attribute Schema	7
		Operation Schema	6
		Property Schema	8
		Event Schema	4
		View Schema	9
B.3	**Message Schema**	Message Schema	2
		Parameter Schema	2
		Formato dos Parâmetros conforme Categoria	7
		Schema for Type Message Schema	3
		TypeParameter Schema	2
B.4	**Sub-esquemas gerais**	Measure Schema	2
		External Schema	2
		Import Schema	3
		MultiLanguage Schema	3
		LanguageUnit Schema	2
		Image Schema	4
		MessageFormat Schema	3

Além disto, são definidas duas tabelas auxiliares:

Tipos Primitivos	Tipos de dados primitivos usados nos esquemas.	5 elementos
Formatos	Formatos utilizados nos esquemas.	13 elementos

C. ESQUEMAS EM XML

Apresentamos aqui uma síntese dos esquemas utilizados na definição do modelo abstrato, mapeados para XML e RDF, conforme definido no capítulo 4, tratados detalhadamente no relatório técnico "Esquemas do Modelo *Anima*", disponível integralmente no endereço:

http://www.nuperc.unifacs.br/santanche/reltecnicos/esqanima.pdf

	Esquema	Arquivo	Descrição	Linhas
C.1	**DTD XML para Workspace Schema**	anima-workspace-schema.dtd	DTD	68
		workspace.xml	Exemplo	87
C.2	**DTD XML para CIM Schema**	anima-cim-schema.dtd	DTD	153
		cim.xml	Exemplo	158
C.3	**RDF Schema para CIM Schema**	anima-basic-schema.rdf	Define os tipos básicos utilizados no esquema CIM. Mapeia a maior parte da "Tabela de Tipos Primitivos" e "Tabela de Formatos" definidas no Apêndice B.	261
		anima-language-schema.rdf	Atua de forma complementar ao anima-basic-schema, definindo tipos associados a linguagens naturais, conforme RFC1766 [ALV95].	42
		anima-general-schema.rdf	Define sub-esquemas gerais utilizados no esquema CIM. Mapeia "Sub-esquemas gerais" definidos no Apêndice B.4.	151
		anima-cim-schema.rdf	Define sub-esquemas gerais utilizados no esquema CIM. Mapeia "Sub-esquemas gerais" definidos no Apêndice B.4.	300
		cim.rdf	Exemplo	262
C.4	**Folha XSLT exemplo XML → Casa Mágica / HTML**	workspace-casa-html.xsl	Converte uma composição representada em XML para uma página HTML	133
		resultado.html	Resultado	92
C.5	**Representação XML para Message Schema**		Representação para as 7 categorias.	

D. FERRAMENTAS EM JAVA PARA SUPORTE AO PROJETO *ANIMA*

A seguir, são apresentadas as bibliotecas e ferramentas que dão suporte à implementação Java de sistemas *Anima*, conforme categorias descritas no tópico 4.4. São relacionadas as classes e ferramentas mais relevantes dentro do conjunto.

D.1. Bibliotecas

Suporte a objetos *Anima*

AnimaInterface	Interface a ser implementada em qualquer componente *Anima*. Define o conjunto mínimo de serviços de que um objeto deve dispor, para se integrar em um ambiente *Anima*, como: serialização/deserialização do estado do objeto e envio/recebimento de mensagens.
AnimaObject	Concentra métodos que dão suporte à serialização/deserialização do estado do objeto, conforme o padrão *Anima*.
AnimaMessage	Concentra métodos que dão suporte à montagem e desmontagem de mensagens, conforme o padrão *Anima*.
Message	Classe que representa genericamente uma mensagem Anima. Todos os demais tipos e categorias de mensagem são herdeiros de Message.

Acesso e manutenção de representação CIM

InfoNode	Nó de uma árvore associada a informações CIM. No primeiro nível podemos ter uma raiz ou uma lista de raízes que representam os recursos descritos. Abaixo de cada recurso se encontra uma lista de InfoNodes, contendo a descrição de suas propriedades e respectivos valores.
InfoRDF	Gera um documento RDF a partir de árvores de InfoNodes
BeanParser	Varre um JavaBean e extrai toda a sua estrutura na forma de uma árvore de InfoNodes.
RDFInfo	Gera árvores de InfoNodes a partir de documentos RDF. Utiliza um parser para extrair o conteúdo do documento RDF.
AnimaCIM	Cada objeto desta classe é uma representação CIM. Esta classe mapeia a estrutura descritiva CIM em um modelo baseado em objetos.

D.2. Utilitários CIM

ExtractRDF	Utilitário executado em linha de comando, que extrai informações (padrão CIM) de um JavaBean e gera um arquivo RDF com as mesmas
ExtractInfo	Utilitário executado em linha de comando, que apresenta na tela do terminal informações referentes ao CIM de uma classe. As informações podem ser extraídas diretamente de um JavaBean ou a partir de um arquivo RDF no padrão CIM.
TestRDF	Utilitário executado em linha de comando, utilizado para testes e verificações relativas à estrutura RDF dos documentos padrão CIM. Ele apresenta um relatório, destacando alguns aspectos da leitura do documento.
BuildCIMClass	Utilitário que gera o esqueleto de uma classe Java (arquivo fonte), conforme padrão Anima, a partir de um arquivo RDF, conforme representação CIM.

E. ESTUDOS DE CASO E APLICAÇÕES PRÁTICAS

Apresentamos aqui uma síntese de um estudo de caso e aplicações práticas, tratados detalhadamente no relatório técnico "Objetos e componentes de *software* educacionais", disponível integralmente no endereço:

http://www.nuperc.unifacs.br/santanche/reltecnicos/objcompeduc.pdf.

E.1. Pacotes de *Software* Educacional – Estudo de Caso

O principal argumento que desejamos explicitar neste estudo está centrado sobre as limitações inerentes à estrutura monolítica de grande parte do *software* educacional produzido, que não possibilita a sua integração.

Com o objetivo de destacar afinidades e recursos complementares entre os pacotes de *software* educacional, escolhemos três deles que se destacam por suas qualidades e por tratar, sob perspectivas diversas, temas no domínio da matemática. Aplicamos cada um deles à solução de um mesmo problema. Foram analisados os seguintes pacotes:

WinPlot	O WinPlot é um software, produzido por Richard Parris, cujo principal objetivo é a plotagem de curvas e superfícies em duas ou três dimensões. Ele possui recursos de animação que permitem explorar diversos aspectos da construção de gráficos (endereço: http://math.exeter.edu/rparris/winplot.html).
Modellus	Modellus é um *software*, através do qual é possível "criar, simular, e analisar modelos interativamente no computador, seja de dados e imagens experimentais ou do pensamento puramente teórico". [TEO00] (endereço: http://phoenix.sce.fct.unl.pt/modellus/).
Homos	Este software é destinado a simulações com propósitos educacionais e tem como base os autômatos celulares. Em cada célula pode ser colocado um objeto, cuja classe é definida pelo usuário.

Como resultado da análise, verifica-se que os pacotes dispõem de recursos complementares e que muitas vezes multiplicam os esforços na solução de um mesmo problema. Sua integração traria um benefício recíproco e potencializaria os resultados, mas sua estrutura monolítica dificulta ou impede a integração.

E.2. Construção de *software* utilizando objetos e componentes

Este estudo de caso analisa a construção de um *software*, a ser utilizado por um professor em uma aula de geometria, sob três abordagens distintas:

- **Tradicional** – consiste em construir um único programa de computador monolítico que desenvolva a atividade pedagógica.
- **Utilizando a Programação Orientada a Objetos**

- **Utilizando Componentes de *Software***

Através do estudo, é possível verificar as vantagens do uso da programação orientada a objetos sobre a programação tradicional. Adicionalmente, verificam-se os benefícios de aliar a tecnologia dos componentes de *software* aos objetos.

E.3. Projetos educacionais baseados em objetos e componentes

Neste tópico, sintetizamos uma análise de relevantes projetos, em diferentes áreas, que fazem uso de objetos e componentes de *software* em atividades de ensino aprendizagem:

E-Slate	O E-Slate permite a construção de micromundos, utilizando uma biblioteca de componentes educacionais, especialmente projetada para que sejam facilmente interligados (endereço: http://e-slate.cti.gr/).
AgentSheets	Este é um sistema destinado à construção de simulações, baseado em uma metáfora de agentes autônomos e animados, que se comportam conforme algoritmos independentes de decisão e ação (endereço http://www.agentsheets.com/).
MathWorlds	"[...] provê uma coleção de componentes de software, incluindo um conjunto de mundos de animação e uma variedade de gráficos. Os atores nos mundos (como um palhaço, ou um pato) movem-se de acordo com funções matemáticas." [ROS98.2] (endereço http://tango.mth.umassd.edu/).
Alice	Dirigido a usuários leigos em computação gráfica, consiste em um ambiente para a construção de animações tridimensionais. Ele é um dos melhores exemplos de como se pode explorar a metáfora de objetos para tornar a interação homem-máquina mais intuitiva (http://www.alice.org).
Squeak	É um sistema moderno que implementa a linguagem Smalltalk. Ele constitui um esforço para o resgate do antigo propósito desta linguagem, de tornar a construção de aplicações no computador acessível a qualquer usuário (endereço http://www.squeak.org).

E.4. Integrando e Colaborando

Neste tópico, sintetizamos uma análise de relevantes projetos que visam a integração e distribuição de componentes de *software* educacional:

ESCOT	Sistemas como o MathWorlds, E-Slate, AgentSheets e JavaSketchpad estão inseridos no projeto ESCOT [ROS98], que tem estudado meios de realizar a integração destes diferentes sistemas, baseando-se no modelo de componentes intercomunicantes (endereço http://www.escot.org).
Squeak	Projeto de integração, no qual um conjunto de objetos produzido no Alice é utilizado no ambiente do Squeak. Deste modo, é possível, através de comandos na linguagem Smalltalk, controlar o coelho tridimensional do Alice.
Bibliotecas de Objetos	Alguns projetos têm investigado mecanismos para a constituição de bibliotecas de objetos e componentes de software para o contexto educacional. Isto inclui uma série de desafios, entre eles definir quais são as características de um objeto ou componente de software apto a ser distribuído por uma biblioteca. Projetos: MERLOT (http://www.merlot.org/) e EOE (http://www.eoe.org/).